(1)

MAINTENANCE METRICS

FOR JOVIAL (J73) SOFTWARE

THESIS

Douglas R. Tindell
Captain, USAF

AFIT/GE/ENG/88D-57

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY
AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT/GE/ENG/88D-57

MAINTENANCE METRICS

FOR JOVIAL (J73) SOFTWARE

THESIS

Douglas R. Tindell
Captain, USAF

AFIT/GE/ENG/88D-57

Approved for public release; distribution unlimited

AFIT/GE/ENG/88D-57

MAINTENANCE METRICS
FOR
JOVIAL (J73) SOFTWARE

THESIS

Presented to the Faculty of the School of Engineering
of the Air Force Institute of Technology
Air University
In Partial Fulfillment of the
Requirements for the Degree of
Master of Science in Electrical Engineering

Douglas R. Tindell, B.E.E., M.P.A.
Captain, USAF

DECEMBER 1988

Preface

The goal of this study was to determine whether metrics could indicate the maintainability of JOVIAL (J73) software, and, if any were found, to implement selected ones.

An extensive literature search was performed to identify metrics that met three criteria: indicated maintainability, possessed empirical support, and applied to JOVIAL (J73). Since maintainability could not be measured directly, a strong indicator, complexity, was used. Based on evaluation criteria and sponsor requirements, the information metric was selected for implementation from several complexity metrics.

The implementation was tested with both simple code examples and actual F-16 flight software. A plan to compare the results of the metric with the results of the sponsor's manual review method was described.

I am indebted to several people for their assistance and encouragement during this effort. Many thanks go to Major Jim Howatt, my thesis advisor, for his kind patience and valuable guidance. A special thanks must go to Major John Stibravy whose willingness to read yet another draft exceeded even my imagination. Professor Dan Reynolds and Doctor Tom Hartrum deserve thanks for their efforts to ensure my work was understandable and correct. I am especially grateful to my sponsor, Captain Dan Telford, for his support and genuine interest. Most importantly, I thank my wife [redacted] for her love and encouragement throughout the AFIT graduate program.

Table of Contents

Volume 1

Volume 2

List of Figures

List of Tables

Abstract

The expense of maintaining software is greater than any other phase in the life cycle. To help reduce the costs, software which may not be maintainable should be identified before being released for use. Measures of software quality, or metrics, may be able to help do this. The goal of this study was to identify measures which could indicate the maintainability of JOVIAL (J73) software, and to implement selected ones.

Maintainability cannot be measured directly, so a strong indicator, complexity, was measured instead. Five categories of complexity metrics were reviewed: size, control, data, information, and hybrid. Through an analysis of metrics from each category, the information metric was selected for implementation.

Using Ada as the implementation language, an analyzer to compute the information metric was constructed. The design was primarily object-oriented but was influenced by compiler theory. The resulting analyzer can easily incorporate new metrics or new input/output requirements.

Testing was performed using both simple code examples and actual F-16 flight software. The analyzer properly computes the information metric with few exceptions. A plan to compare the results of the analyzer with the results of the sponsor's present manual review was described.

MAINTENANCE METRICS FOR JOVIAL (J73) SOFTWARE

I. Introduction

Software is a critical area of concern in today's U. S. Air Force. The primary reason is a large and growing list of weapon systems that depend directly on embedded computers. Some of the systems included in this list are: B1-B, AMRAAM, LANTIRN, F-111, F-15E, F-16, Peacekeeper, and Minuteman III. In each weapon system, one or more computers are used, and each computer must have software to work. The performance of these systems depends, in a very real sense, on the software operating within the embedded computers. "Software has become the pacing technology in advanced fighters, just as it has in most other weapon systems and information systems" (Canan, 1986:49). Another reason for concern is the amount of software used by each system. Just as the number of computers used in a weapon system is increasing, the amount of software used by a single computer is increasing. The reason for this is twofold. First, the capabilities of weapon systems are being expanded. As the operational requirements become more complex, so must the software. Second, more functions are being done in software rather than in hardware. Consideration of these facts will lead to a better understanding of the significance of software in today's weapon systems:

"- Through most of the Vietnam War, Air Force F-4's contained no digital computers and no software.

- Each F-16A that went operational in 1981 had seven computer systems with fifty digital processors and 135,000 lines of code.

- This year's F-16D has fifteen computer systems with 300 digital processors and 236,000 lines of code.

- Moreover, the magnitude of the software inside an aircraft may represent only a fraction of that aircraft's total software requirement." (Canan, 1986:49)

One more reason for concern is a definite lack of software professionals (programmers and managers). The claim that "creating software is still much more an art than a science" marks program development as a largely labor-intensive, human endeavor (Canan, 1986:50). Individual effort and creativity therefore play a crucial role in developing software. It also means that automated tools and development environments, while helpful, can't boost software productivity alone. Art or science, not enough trained people exist to fill the need. A final reason for concern is the effort required to keep a fielded weapon system up-to-date. As targets, equipment, and user's needs change, so must the software. This can be very expensive. By 1990, ten percent of the total defense budget is predicted to be spent on software alone. For example,

> it costs $85 million to develop the software for an F-16D. It costs another $250 million to maintain that software--rectifying its errors, keeping it in shape, updating it--over its anticipated operational lifetime [Canan, 1986:49].

The bottom line: availability of software for current and future weapon systems is restricted by a large and increasing

demand, greater complexity, low productivity, and huge maintenance costs. (Canan, 1986:46-52)

Background

An ever-increasing level of computational hardware sophistication coupled with the need to produce and operate the associated software has prompted several authors to declare that a "software crisis" exists (Pressman, 1987:13; Conte and others, 1986:1; Canan, 1986:46; Lehman, 1985:491). This "crisis" is not a simple situation where demand for software greatly exceeds the ability to produce. It is much more involved.

> Rather, the software crisis encompasses problems associated with how we develop software, how we maintain a growing volume of existing software, and how we can expect to keep pace with a growing demand for more software [Pressman, 1987:13].

Thus, the software crisis actually extends over the entire software development life cycle. Although different development models have been proposed, most establish several independent phases with defined start and stop points. For this study, these phases may be grouped into four categories which encompass these basic concepts:

- Design
- Coding
- Testing
- Maintenance

(Conte and others, 1986:6)

Although software design, coding, and testing have been discussed at length in the literature, relatively little

emphasis has been placed on maintenance (Swanson, 1976:492; Parikh, 1982:10). As an illustration, one author proclaimed that "to work in maintenance has been akin to having bad breath" (Schneidewind, 1986:303). Despite this opinion of software maintenance, the truth is that maintenance costs may range from 40 to 90 percent of the total cost of a large-scale software system (Boehm, 1973:48; Zelkowitz, 1978:202; Harrison and others, 1982:65; Pressman, 1987:50, 525). Boehm was one of the first researchers to recognize this. In an article entitled "Software Engineering" he said,

> despite its size, software maintenance is a highly neglected activity. There are a few good general principles and few studies of the process, most of them inconclusive [Boehm, 1976:1236].

Even though the majority of life cycle resources are expended in the maintenance phase, it is the most ignored phase!

Several possible reasons for this paradox are:

"- 75-80 percent of existing software was produced prior to significant use of structured programming

- It is difficult to determine whether a change in code will affect something [i.e., some other portion of the code]

- It is difficult to relate specific programming actions to specific code" (Schneidewind, 1987:303)

Essentially, software cannot be maintained unless it was designed to be maintained (Schneidewind, 1987:303). Also, the misconception that software maintenance holds no creative challenge is a severe detriment to attracting talented professionals to the field (Liu, 1982:25). Perhaps that is

why "in general, less-qualified personnel are assigned to maintenance tasks" (Boehm, 1976:1236).

Boehm postulated that common software development practices may be at the root of the problem. Cost and schedule often drive the software effort. Another prime consideration is hardware optimization. Special emphasis of any of these areas will lead to increased maintenance costs for the software produced. (Boehm, 1976:1236)

Modern software systems must emphasize maintainability to keep the overall system life cycle costs down to manageable levels.

> The major expenses in computer systems at present are in software. While the cost of hardware is decreasing rapidly, software productivity improves only slowly. Thus, the cost of software relative to hardware is rapidly increasing. The majority of this software cost can be attributed to software maintenance [Yau and Collofello, 1985:849].

Long-term system software costs skyrocket when performance of maintenance tasks is difficult. A software system that is hard to maintain is an expensive one. The key to holding down the cost of a large-scale software system, then, is to write the software so that maintenance is easy (Gustafson and others, 1985:2).

The phrase "software maintenance" is misleading. (The Air Force has even dropped the term "maintenance" and refers instead to "post-deployment software support".) Intuitively, maintenance means the replacement of damaged or worn components to return the affected system to a former state. Since software does not wear out or break in the traditional

sense, "software maintenance" means that current program execution is considered unacceptable and will be changed. According to Lehman and Belady, reasons for unacceptability include: correction of errors, incorporation of improvements, or adaptation to new environments. Yau and Collofello agree. They state that "software maintenance is a very broad activity that includes error corrections, enhancements of capabilities, deletion of obsolete capabilities, and optimization" (Yau and Collofello, 1985:850). Perhaps the best statement of the activities involved in software maintenance was given by Swanson. He (Swanson, 1976:492-494) divided maintenance activities into three categories:

> Corrective Maintenance - Activities performed in response to either a processing failure (software doesn't work), performance failure (software doesn't work right), or implementation failure (software doesn't meet implementation standards).
>
> Adaptive Maintenance - Activities performed in response to changes in data or processing environments.
>
> Perfective Maintenance - Activities performed "to eliminate processing inefficiencies, enhance performance, or improve maintenance" (Swanson, 1976:493).

Note the implication is that software maintenance somehow changes the software away from its original state to a new state. This is entirely different than the intuitive meaning of "maintenance." (Belady and Lehman, 1985:304)

Once the basic concepts behind the maintenance activity have been stated, a formal definition can be made. Most software maintenance researchers have given their own

definition that emphasizes particular points they consider important to the activity, and brief a look at some of the definitions used is helpful. Software maintenance is:

> The job of correcting errors and changing program operation as requirements change [Berns, 1984:14].
>
> The process of "fixing" the system so that its operation more closely corresponds to the desired characteristics [Conn, 1980:401].
>
> A continuing phase in which additional discovered errors are corrected, changes in code and manuals are made, new functions are added, and old functions deleted [Conte and others, 1986:4].
>
> The action taken under stated conditions to restore a failed system to operable condition within a specified time. (Gilb, 1977:26)
>
> The ease with which software can be understood, corrected, adapted, and/or enhanced [Pressman, 1987:531].
>
> Modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a changed environment [Schneidewind, 1987:309 who referenced the ANSI/IEEE Standard 729, 1983].

All of these definitions are very similar, and do encompass the three categories of activity given by Swanson. Thus, any one may be used for this research effort.

Problem

This study's sponsor, an Air Force Operational Test and Evaluation Center (AFOTEC) unit responsible for reviewing F-16 flight software releases, wants to improve its ability to identify software that may be difficult to maintain. A manual review process is currently used, but the procedures and tools available do not provide the ability to thoroughly

review even 10 percent of the total code to be analyzed (Telford, 1988). Therefore, the use of automated software metrics has been proposed to augment it.

Scope

This research studied the difficulty of evaluating effectively the maintainability of software written in the JOVIAL (J73) programming language, and was concerned only with the following:

- Quality factors which apply to software maintenance
- Metrics which measure these factors and have empirical data available to back up their applicability to maintenance
- Implementation of these metrics for JOVIAL (J73) software

Limitations

No new metrics were proposed. The goal of this study was not to propose new maintenance metrics but rather to research whether existing metrics could be used to predict the maintainability of JOVIAL (J73) software.

Any JOVIAL (J73) software used for analysis was assumed to be syntactically valid.

Approach

Based on the scope of this study, a thorough review of software metrics literature was absolutely necessary. This provided the theoretical background required to identify maintenance quality factors and to select appropriate metrics

to measure them. As many of the selected metrics as possible were implemented.

To implement the selected metrics as simply as possible, an effort to adapt existing code from the JOVIAL Automated Verification System (JAVS) and the Integrated Tool Set (ITS) JOVIAL (J73) compiler was made. Since neither approach was feasible, new code using the JANUS/Ada (trademark of R. R. Software) programming language was written.

Comparison of the expected results of the metrics with the results of the present manual process would normally complete this kind of research. However, the results of the sponsor's manual review process were available too late to be included in this thesis. Accordingly, a plan describing how the results could be compared was included for future study.

Thesis Organization

Chapter 2 reviews the current literature with emphasis on maintenance quality factors and associated metrics that have empirical evidence to support them. The metrics that were implemented in this study are selected, justified, and refined in Chapter 3. The design of the analyzer which implements the selected metrics is documented in Chapter 4. Chapter 5 contains the analysis of some JOVIAL (J73) software using the analyzer. Conclusions about this research effort and recommendations for future study are presented in Chapter 6.

II. Literature Review

Chapter 1 revealed that even though software maintenance demands more resources than any other development phase little effort to find ways to minimize the required resources has been made. However, the spiraling expense of maintaining software systems which may remain in service for 10 to 20 years has begun to attract the attention of researchers (Harrison and others, 1982:65; Kearney and others, 1986:1044; Kafura and Reddy, 1987:335).

As research has progressed, improving the "quality" of software has often been touted as the overall goal. But just what is "software quality"? It could be described as "a distinquishing attribute which indicates a degree of excellence" (McCall and others, 1977:2-1). Clearly, such a hazy description is not much help in developing techniques to produce quality software, but it may be the best definition possible (Mohanty, 1979:251).

Quality

The idea of measuring the quality of software has been around since at least 1968. In a short paper published that year, the authors stated why they felt measures of software quality were needed.

> In the absence of specific, applicable quantitative measurement tools there exists no means of defining the desired level of quality in a computer program, where quality is considered as something beyond correct program functioning, nor of ascertaining whether the desired level has been achieved [Rubey and Hartwick, 1968:671].

Quality was considered to be composed of a number of "quality attributes" where an attribute was defined to be "a precise statement of a specific software characteristic" (Rubey and Hartwick, 1968:671). These attributes linked together to form a hierarchy resembling a pyramid. At the top was high-quality software and at the bottom was a large number of minor attributes. A few major attributes lay in the middle. These were broad enough to specify the qualities desired, yet focused enough to support quantitative measurements. In all, seven groups of major attributes were given.

> "A_1 - Mathematical calculations are correct[ly] performed.
>
> A_2 - The program is logically correct.
>
> A_3 - There is no interference between program entities.
>
> A_4 - Computation time and memory usage are optimized.
>
> A_5 - The program is intelligible.
>
> A_6 - The program is easy to modify.
>
> A_7 - The program is easy to learn and use." (Rubey and Hartwick, 1968:672)

Each group contained a number of sub-attributes which did not have to be applied equally to a program. Depending on the specific software, some sub-attributes would have greater importance than others. (Rubey and Hartwick, 1968:671-672)

The next major effort to define software quality was made by Tom Gilb in his book, Software Metrics. He focused attention on developing practical techniques for measuring software quality, even though most of the metrics he proposed are impractical to implement. In Gilb's words,

> All critical software concepts have at least one practical way of being measured. It may not be elegant, but if it improves results, it should be used until perfectionists can indicate a better tool [Gilb, 1977:16].

At about the same time Gilb's book was published, two other works dealing with software quality emerged. Both proposed a series of software measures, but unlike Gilb the authors developed models of software quality to support their proposed quantitative measures.

Models

When details are suppressed to permit underlying properties to become visible, a model is being used. Thus, a model

> is simply an abstraction of a real world process or product. It attempts to explain what is going on by making assumptions and simplifying the environment [Basili, 1985:3].

The activity of conceiving, shaping, and refining the software is the process, and the completed software produced by the process is the product (Conte and others, 1986:20-21). Two kinds of models may be constructed. An empirical model is one that is made by fitting equations that express the desired information to observed data. Validation of the model may be performed through correlation of the equations and the data. An analytic model is one that is made under the assumption that the process or product being modeled is well-enough understood that equations to express the desired information can be written beforehand. Validation of this

model is accomplished by "fine tuning" the equations as observed data is reviewed. (Shaw, 1981:255-256)

One of the earliest efforts to define a software model was made by Boehm and others. Using intuition and logic, they listed and organized a number of characteristics that described software quality. Initially, a working set of eleven characteristics was defined. Later, the set was revised to eliminate overlap, and a hierarchy of order applied. A set of twelve "primitive characteristics" that related to seven higher characteristics were formed (Boehm and others, 1978:3-18). These seven were grouped under two higher characteristics which in turn were grouped under a single overall characteristic. Figure 1 illustrates the resulting structure. The higher characteristics were thought to relate more closely to the user while the lower ones were intended to make it easier to define quantifiable measures. Table 1 shows the characteristic descriptions. (Boehm and others, 1978:x-xiv, 1-1 thru 1-5, 3-1 thru 3-26)

Another early effort to define a software quality model was made by McCall and others. Working under an Air Force contract managed by the Rome Air Development Center (RADC), this group took much the same approach as Boehm and others but defined a larger number of characteristics. Their first task was a literature search to identify characteristics or software quality factors as they called them. This formed the initial list of factors. Next, this list was reviewed to eliminate redundancy and emphasize a user orientation. It

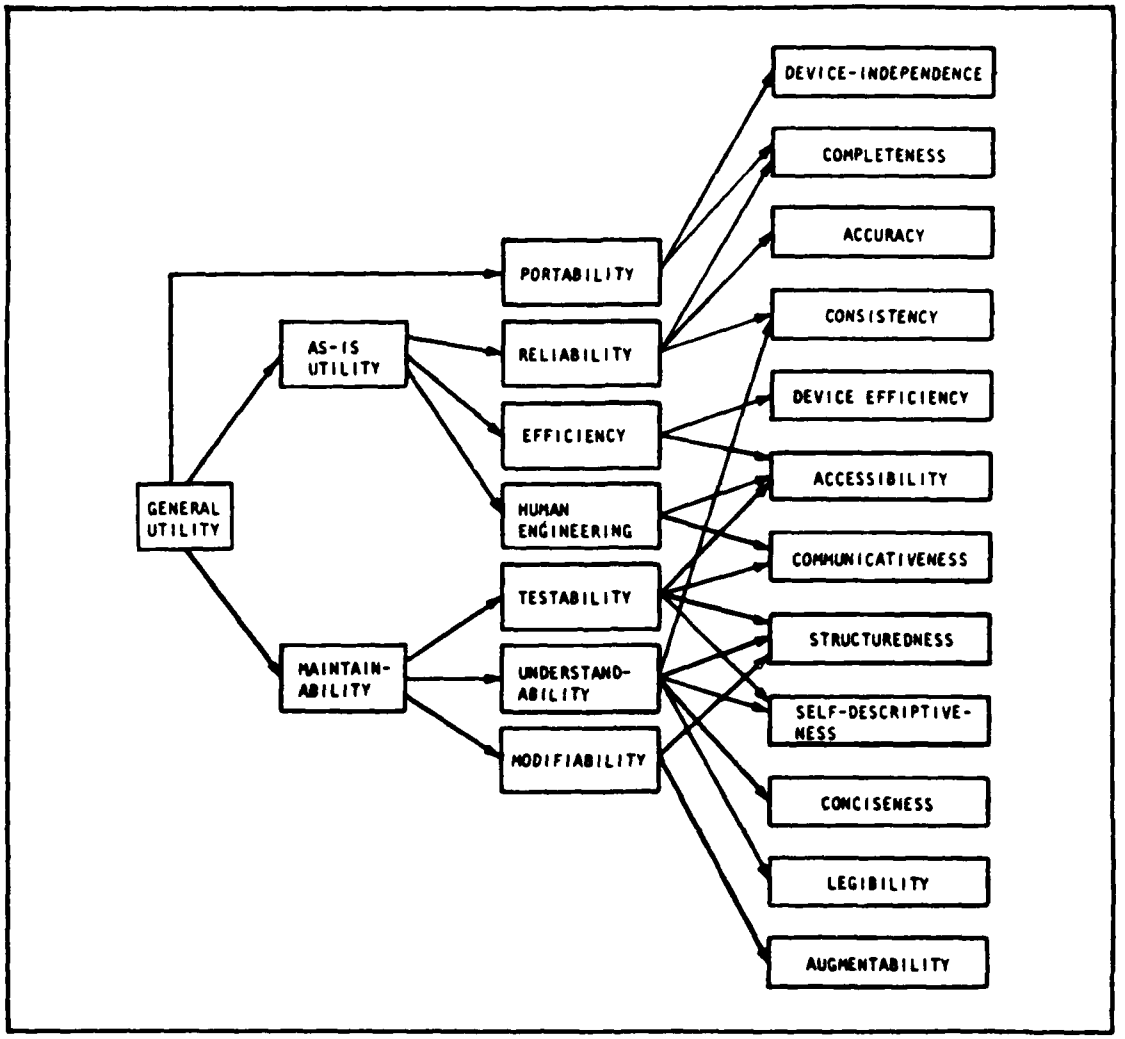

Figure 1. Characteristics Tree (Boehm and others, 1978:xiii)

was during this review process that three different views of the factors became visible: Product Operation, Product Revision, and Product Transition (see Figure 2). These views reflected a strong user viewpoint so they were identified as the top of the software characteristics hierarchy. Each was formed by the contribution of a number of factors and each factor was composed of a number of sub-factors or criteria. McCall, just like Boehm, intended the hierarchy to form a

Table 1

Characteristic Descriptions

Characteristic	Description
General Utility	The extent to which the software can be used on both its current computer system and other computer systems.
As-Is Utility	The extent to which the software can be used in its current form and on its current computer system.
Maintainability	The extent to which the software can be easily changed to meet new requirements.
Portability	The extent to which the software can be run on more than one computer system.
Reliability	The extent to which the software can be expected to satisfactorily perform its intended function.
Efficiency	The extent to which the software can be expected to perform its function without using excessive resources.
Human Engineering	The extent to which the software performs its function without wasting user resources or causing frustration.
Testability	The extent to which the software lends itself to evaluation of acceptance criteria.
Understandability	The extent to which the purpose of the software is clear to the evaluator.
Modifiability	The extent to which the software can be easily changed.
Device Independence	The extent to which the software can be run on computer systems other than its current one.

- continued -

Table 1 - continued

Characteristic Descriptions

Characteristic	Description
Completeness	The extent to which all of the software components exist and are fully developed.
Accuracy	The extent to which the output of the software satisfies the intended purpose.
Consistency	The extent to which the software contains uniform nomenclature.
Device Efficiency	The extent to which the software uses machine resources without excessive waste.
Accessibility	The extent to which selected software components can be easily used.
Communicativeness	The extent to which easily-understood, useful inputs and outputs are provided.
Structuredness	The extent to which a logical organization exists between software components.
Self-Descriptiveness	The extent to which the software contains information to allow a reader to understand its purpose, scope, limitations, organization, inputs, and outputs.
Conciseness	The extent to which the software contains only necessary information.
Legibility	The extent to which the purpose of the software components as well as the overall purpose is easily understood by reading the code.
Augmentability	The extent to which the software can be easily changed to encompass additional data structures or computational functions.

(Boehm and others, 1978:3-1 thru 3-26)

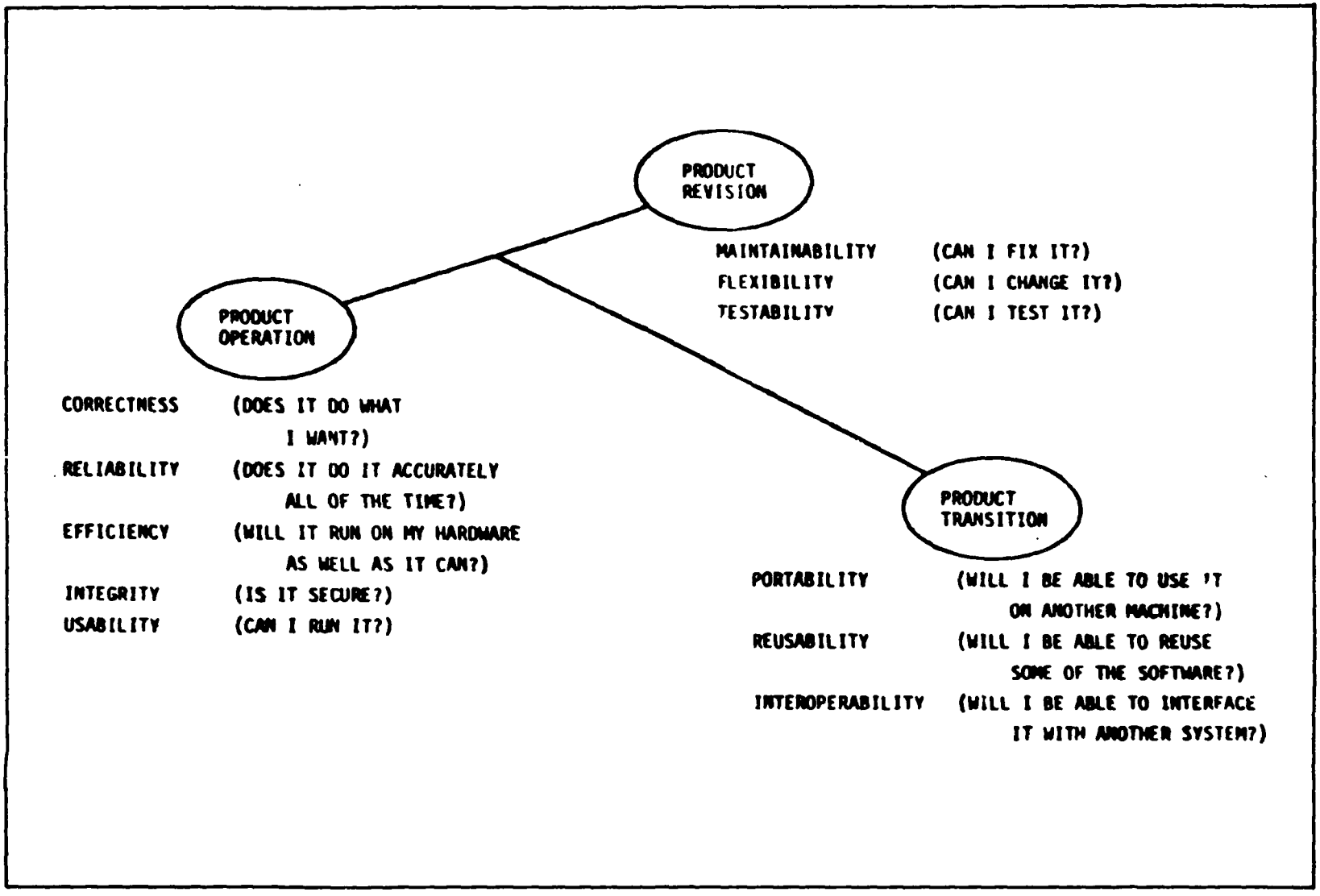

Figure 2. Three Views of Software Quality (McCall and others, 1977:3-2)

dichotomy with user-oriented characteristics at one end and measurement-oriented characteristics at the other. The final hierarchy is made up of the three views at the top, eleven factors in the middle, and twenty-five criteria at the bottom. Table 2 lists the description of each view, factor, and criteria. (McCall and others, 1977:2-1 thru 4-11)

Curtis recognized the similarity of these approaches for quantitatively measuring software quality. To emphasize the parallel ideas embodied in the Boehm and others model and the McCall and others model, he integrated the structure of both into a single diagram. See Figure 3. User-oriented characteristics lie at either end of the diagram. Progress inward from either end of the diagram reveals a middle layer of in-between characteristics and finally a bottom layer of measurement-oriented characteristics. The actual software measures, located in the center, are labeled "metrics."

Both software quality models provide a theoretical basis for measuring the degree to which certain characteristics are exhibited by the software being analyzed. However, it must be realized that "the desireable qualities of a software product vary with the needs and priorities of the prospective user" (Boehm and others, 1980:220).

The interaction of software with people is critical to software maintainability. This software-human relationship is frequently approached from the viewpoint of software quality (Baker and Zweben, 1980:506). In this effort, the maintainability of software is the quality being studied.

Table 2

Software Quality Factor Definitions

View	Definition
Product Operation	The user's need for software that can be operated easily and that performs its function well.
Product Revision	The user's need for software that can be easily changed to fix problems and meet new requirements.
Product Transition	The user's need for software that is easy to use with and on other computer systems.
Factor	**Definition**
"Usability	The effort required to learn, operate, prepare input, and interpret output of a program.
Integrity	The extent to which access to software or data by unauthorized persons can be controlled.
Efficiency	The amount of computing resources and code required by a program to perform a function.
Correctness	The extent to which a program satisfies its specifications and fulfills the user's mission objectives.
Reliability	The extent to which a program can be expected to perform its intended function with required precision.
Maintainability	The effort required to locate and fix an error in an operational program.
Testability	The effort required to test a program to insure it performs its intended function.

- continued -

Table 2 - continued

Software Quality Factor Definitions

Factor	Definition
Flexibility	The effort required to modify an operational program.
Reusability	The extent to which a program can be used in other applications - related to the packaging and scope of the functions that programs perform.
Portability	The effort required to transfer a program from one hardware configuration and/or software system environment to another.
Interoperability	The effort required to couple one system with another." (McCall and others, 1977:3-5)
Criteria	**Definition**
Operability	Those attributes of the software that provide for minimum processing time.
Training	Those attributes of the software that provide for control of the access of software and data.
Communicativeness	Those attributes of the software that provide useful inputs and outputs which can be assimilated.
I/O Volume	The amount of information that can be input and output through the software.
I/O Rate	The speed at which the information can be passed by the software.
Access Control	Those attributes of the software that provide for control of the access of software and data.
Access Audit	Those attributes of the software that provide for an audit of the access of software and data.

- continued -

Table 2 - continued

Software Quality Factor Definitions

Criteria	Definition
Storage Efficiency	Those attributes of the software that provide for minimum storage requirements during operation.
Execution Efficiency	Those attributes of the software that provide for minimum processing time.
Traceability	Those attributes of the software that provide a thread from the requirements to the implementation with respect to the specific development and operational environment.
Completeness	Those attributes of the software that provide full implementation of the functions required.
Accuracy	Those attributes of the software that provide the required precision in calculations and outputs.
Errcr Tolerance	Those attributes of the software that provide ccntinuity of operation under nonnominal conditions.
Consistency	Those attributes of the software that provide uniform design and implementation techniques and notaticn.
Simplicity	Those attributes of the software that provide implementation of functions in the most understandable manner. (Usually avoidance of practices which increase complexity.)
Conciseness	Those attributes of the software that provide for implementation of a function with a minimum amount of code.

- continued -

Table 2 - continued

Software Quality Factor Definitions

Criteria	Definition
Instrumentation	Those attributes of the software that provide for the measurement of usage or identification of errors.
Expandability	Those attributes of the software that provide for expansion of data storage requirements or computational functions.
Generality	Those attributes of the software that provide breadth to the functions performed.
Self-Descriptiveness	Those attributes of the software that provide explanation of the implementation of a function.
Modularity	Those attributes of the software that provide a structure of highly independent modules.
Machine Independence	Those attributes of the software that determine its dependency on the hardware system.
Software System Independence	Those attributes of the software that determine its dependency on the software environment (operating systems, utilities, input/output routines, etc.)
Communications Commonality	Those attributes of the software that provide the use of standard protocols and interface routines.
Data Commonality	Those attributes of the software that provide the use of standard data representations.

(McCall and others, 1977:3-1 thru 3-2, 3-5, 4-4 thru 4-5)

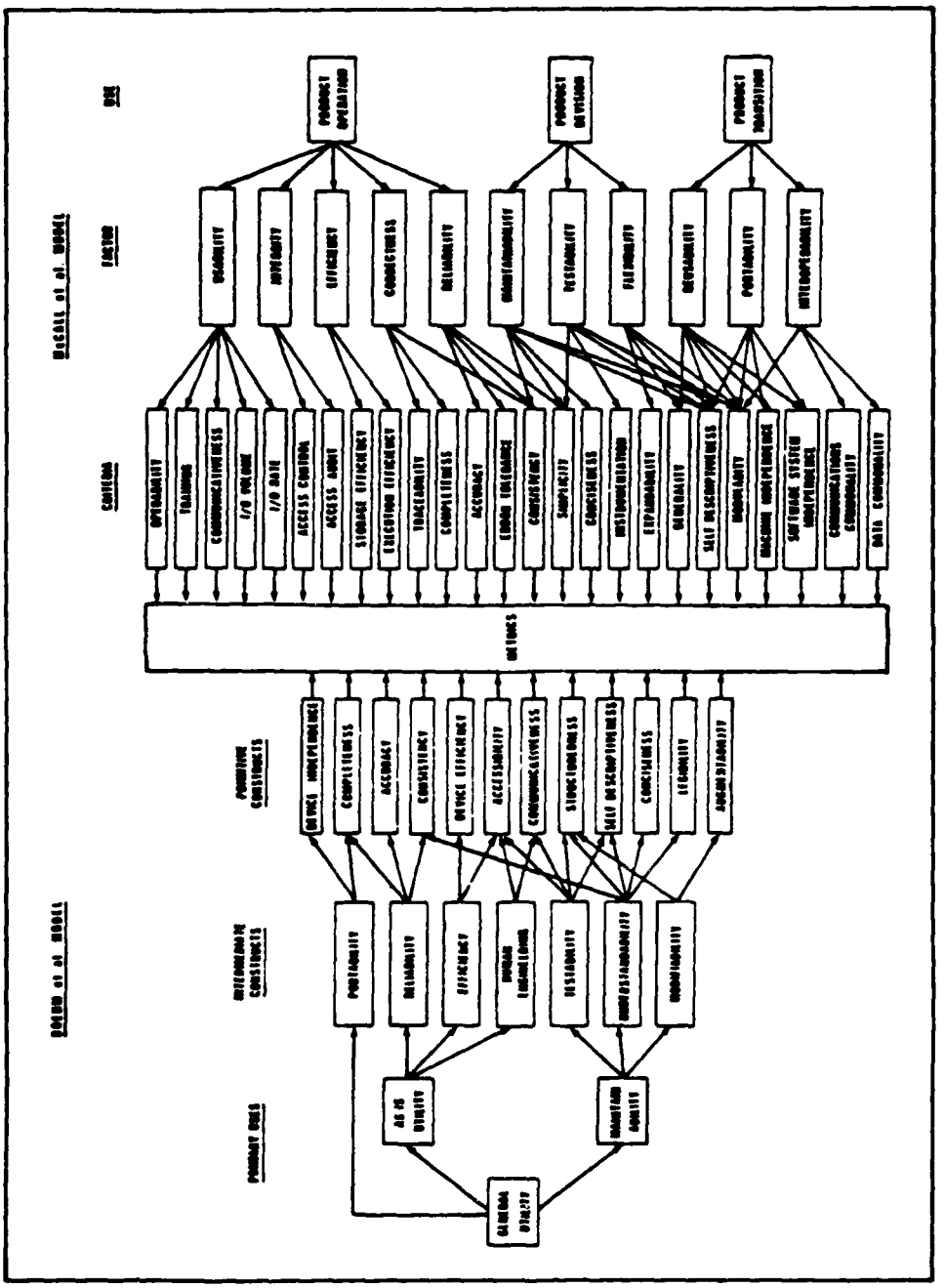

Figure 3. Software Quality Models (Curtis, 1981:206)

Although many characteristics of software maintainability may be identified, complexity is generally felt to be the most important (Bowen, 1978:149).

Complexity

Complexity is considered to be a strong indicator of the maintainability of software (Ivan and others, 1987:94). One expects a more complex piece of software to be more likely to contain errors (Woodward and others, 1979:101; Curtis and others, 1979b:359). Studies indicate that the inability to control changes made to the software after it has been delivered to the user is partially to blame for the high cost of maintenance (Freedman and Weinberg, 1982:53-54; Harrison and others, 1986:65; Kafura and Reddy, 1987:335). This lack of control can be attributed to two reasons:

- Maintenance is performed by a diverse and changing group of people.
- The maintenance phase occurs over a longer period of time so different methodologies may in fact be used. (Kafura and Reddy, 1987:335)

Changes during the design, coding, and testing phases are usually tightly controlled by a single group of people. This is possible since these phases are fixed and relatively short. The time span of the maintenance phase, however, is not limited. The long service lives of some software systems means the likelihood of the same group of people performing maintenance is very small. As maintainers change, so will the way maintenance is performed. Not only do the actual changes to the software differ, the methodology guiding the

maintenance will differ. This is due to either new personal preferences or new ideas about how to perform software maintenance.

As time passes and more changes are made, the software becomes increasingly complex (Harrison and others, 1982:65; Kafura and Reddy, 1987:335). In turn, future maintenance becomes increasingly difficult.

This complexity/maintenance cycle is self-perpetuating. As maintenance is performed, complexity increases (Belady and Lehman, 1985:304). As the complexity increases, the likelihood of needing maintenance as well as the cost of that maintenance increases (Lehman, 1985:494).

If complexity could be quantitatively measured, it would be possible to change maintenance activities to break this vicious cycle. Many researchers feel that this approach holds promise in slowing the increasing cost of software maintenance (Weissman, 1974:25; Harrison and others, 1982:65; Kearney and others, 1986:1044; Kafura and Reddy, 1987:335).

Many studies discuss the relationship between software complexity and maintainability but never bother to define what complexity really is (Curtis, 1981:207-208). Often, it is treated as a "fuzzy feeling" and the reader is expected to understand without explanation (Curtis, 1979:96). However, at least two definitions are available.

> Complexity is a characteristic of the software interface which influences the resources another system will expend while interacting with the software [Curtis, 1979:102].

> Complexity is the measure of the resources expended by another system in interacting with a piece of software [Basili, 1980:232].

These definitions are similar. They concentrate on the interaction of the software with another system which may be either machine or human (Curtis, 1979:102; Basili, 1980:232).

Two broad categories of complexity may be identified: computational and psychological. Computational complexity refers to "...the quantitative aspects of the solutions to computational problems" (Rabin, 1977:625). The hardware-software interface is emphasized. Verification of the efficiency of an algorithm is an example of computational complexity. Psychological complexity refers to the "characteristics of software which make it difficult to understand and work with" (Curtis and others, 1979a:96). The programmer-software interface is emphasized. Determining the difficulty of comprehending and changing a software module is an example of psychological complexity. (Curtis, 1979:95)

At this point, a brief review using the technique of concept mapping (originated by Novak and Gowin, Learning How to Learn, 1984:15-54) is helpful. Figure 4 displays a concept map of this chapter. Boldface words in bubbles represent the concepts which are linked by the lines and words between the bubbles. Generally, software exhibits a number of qualities. Maintainability is primarily driven by complexity so that path is continued. Complexity may be classified as being either computational or psychological. Since software is maintained by people, the path continues

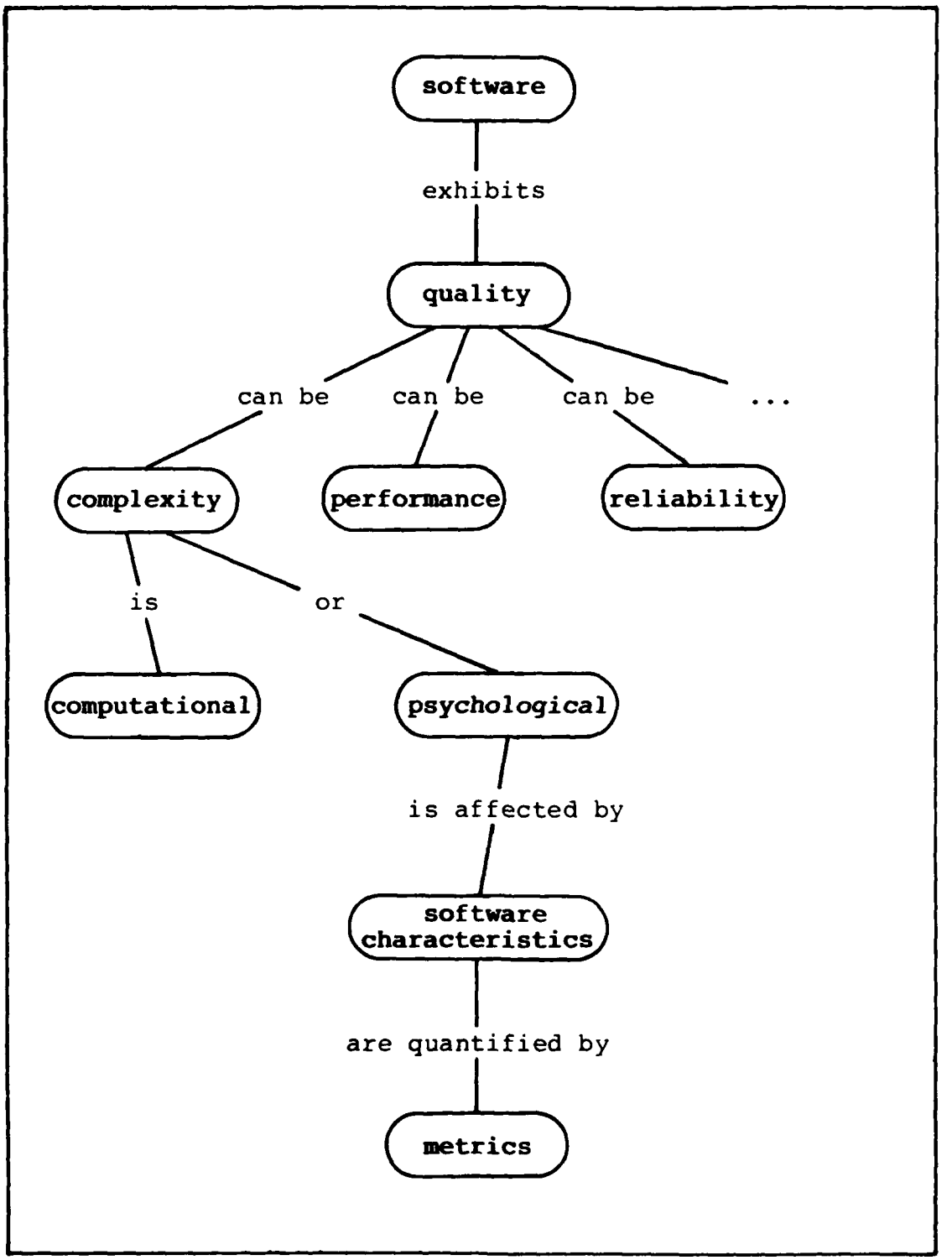

Figure 4. Concept Map of Chapter 2

through psychological complexity. The characteristics of the software directly affect this complexity, and may be quantified by metrics. Figure 5 shows these characteristics and the related metrics which were studied in this research effort. The remainder of this chapter focuses on these complexity metrics.

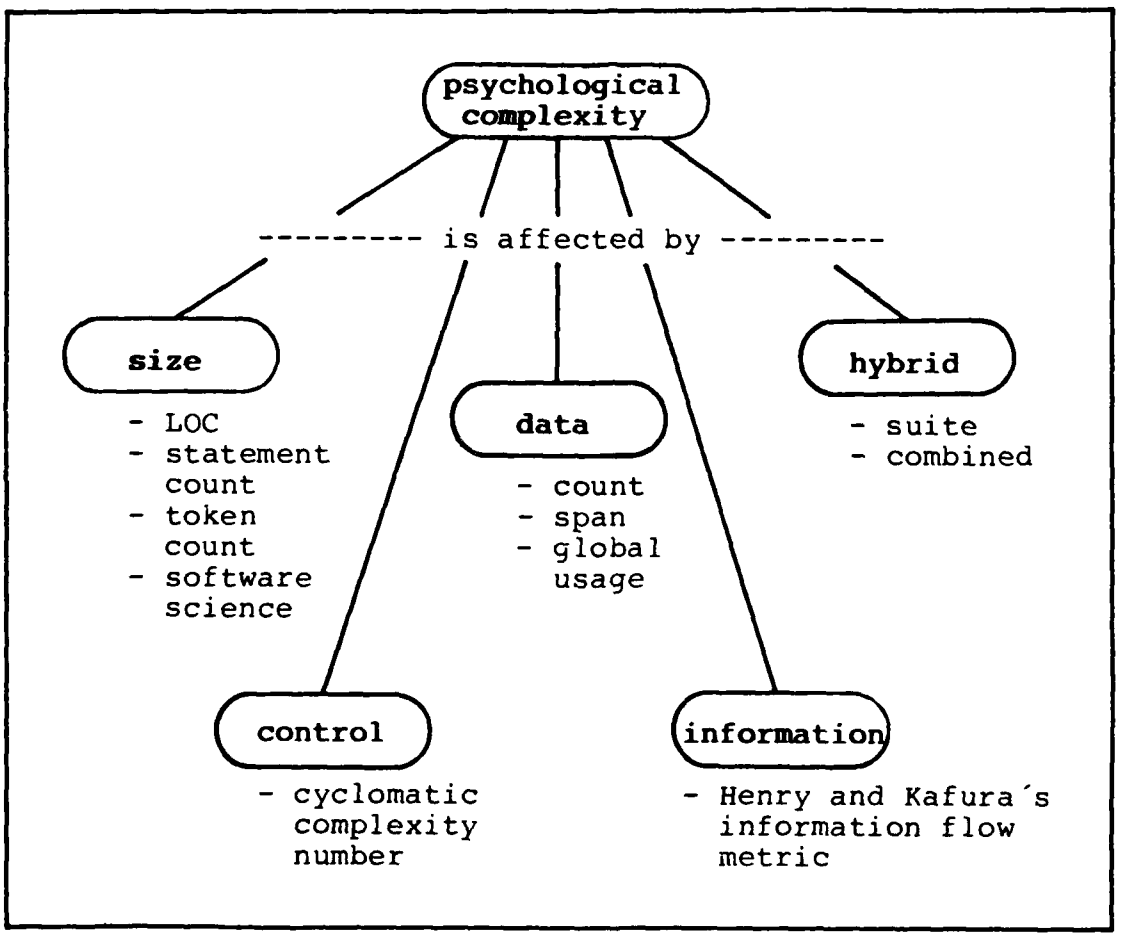

Figure 5. Concept Map of Metrics Studied

Metrics

Techniques for measuring software quality are commonly referred to as metrics. As Gilb points out, "metrics provides us with a powerful language for describing the

relationships which we desire, expect or experience in any grcup of subsystems" (Gilb, 1977:132). The term "complexity metrics" further refines the nature of these measures since they are usually "developed to assess the complexity of programming tasks" (Curtis and others, 1979a:96). Howatt recognized that the term is inaccurate since most measures reference the interaction between program modules rather than between the programmer and program. He also noted that the word "metric" typically refers to a mathematical function between several objects, but is most often used to refer to only one object, the program being measured (Howatt, 1985:1-2). Despite these inconsistencies, this study will use the phrase "complexity metric" to refer to any technique that measures a desired quality of software since this is generally understood.

A metric must be well-defined and consistent if it is to be a useful measure (Levitin, 1986:314; Basili, 1980:232). It should be fairly easy to understand and produce reasonable results. Minor changes in the software should not cause drastic changes in the metric computed.

Of course, "the most desirable metric is a direct measurement" of the quality being measured but that may not be possible for a variety of reasons (Shaw, 1981:252):

- not available when desired
- too expensiv· :o collect
- not possible to make a direct measurement (Shaw, 1981:252)

In the case of maintenance, a direct measurement cannot be made so indirect measurements based on a complexity model must be used. A good complexity metric is one which will:

- relate intuitively to the psychological complexity of the software.
- serve as a tool to compare different software versions.
- show preference to programs written with good programming style.
- be orthogonal to (independent of) other metrics. (Hansen, 1978:29)

Size metrics are measures based on the premise that software becomes more complex as its size increases (Elshoff, 1976:115-116). Included in this category are: lines of code (LOC), statement count, token count, and Halstead's Software Science length and volume. This category is the oldest and has been studied most. Experimental evidence indicates that a larger software system will require more maintenance than a smaller one (Harrison and others, 1982:66; Li and Cheung, 1987:707).

Counting the lines of code is the most common method of measuring software size (Harrison and others, 1982:66). At first brush, it seems relatively simple to count the number of lines in a program. However, not everyone agrees on what constitutes a line of code or even which lines should be counted (some people prefer to exclude comments and data declarations, for example). Despite these shortcomings, LOC remains a popular metric due to its simplicity and ease of computation. (Levitin, 1986:314-315)

Statement count is a measure that originated with the advent of free form programming languages. Languages such as FORTRAN and BASIC can have only one statement per line of code, so statement count is identical to LOC. As free format languages were developed, this was no longer true because statements could be continued past a single physical line or multiple statements could be placed together on one line. Since different languages define a statement differently (some, like Prolog, don't support the idea of a statement), this metric is often not simple to compute. Other problems such as nested statements and statement separators make this measure a tough one to use. (Levitin, 1986:315)

Token count is a metric that is applicable to both fixed *and free form languages.* Indeed, any programming language source code will be broken into tokens in the compilation process. As the basic syntactic unit of software, tokens represent the smallest logical entity which can be measured The single disadvantage token count seems to have is that a large software system will have an enormous number of tokens. (Levitin, 1986:315-316)

In 1972, Maurice Halstead developed a theory of software measurement he called "Software Science" (so called because these measures were supposedly analogous to the laws of the physical sciences). Software Science is based on counts of operands and operators. Although the definition of operands and operators is loose, operands are generally variables, constants, and labels while operators are the arithmetic and

logical symbols which "operate on" the operands. Four counts used in each measure are:

"n_1 = number of unique operators

n_2 = number of unique operands

N_1 = total number of operator occurrences

N_2 = total number of operand occurrences"
(Levitin, 1986:316)

Two size measures, length and volume, have been proven in empirical tests to be useful. They are defined as:

Length $N = N_1 + N_2$

which is all the symbols in a program

Volume $V = N * \log_2 (n_1 + n_2)$

which is the fewest bits needed to encode a program
(Lecciso and others, 1986:606)

Although studies have shown some of the Software Science measures like length and volume to actually measure what Halstead intended (Henry and others, 1981:144), many of his underlying concepts have been criticized as being based on unfounded theory. Apparently, Halstead applied concepts from both computer science and psychology but made assumptions which cannot be totally justified. (Levitin, 1986:316-317; Lecciso and others, 1986:605-608; Curtis and others, 1979a:96; Curtis, 1979:98-99)

Control metrics are measures based on the premise that software becomes more complex as the number of decision points increases (Henry and others, 1981:144). Another way of looking at this is to consider the metric a count of the number of basic control paths through a program. This

explains why control metrics are also known as control flow metrics since program logic may be visualized as "flowing" along the branches in the program. Although many control metrics have been proposed, the one presented by Thomas McCabe in 1976 has certainly received the most attention. (Curtis, 1979:99)

McCabe called his metric the "Cyclomatic Complexity Number," V(G). This number is derived by first constructing the flow graph of the software being measured. A flow graph is simply a directed graph containing a set of nodes and edges. An edge represents a branch in the program control logic while a node represents a section of program statements which are sequentially executed (i.e., not branched from or to). Once the edges and nodes have been identified and counted, the Cyclomatic Complexity Number may be computed using this formula:

V(G) = # of edges - # of nodes + 2p

where p represents the number of connected components in the flow graph
(Curtis, 1979:99)

Several researchers (Henry and others, 1981:148; Curtis, 1981:210-211; Curtis and others, 1979a:102-103; Curtis and others, 1979b:358-359) have studied this metric and reported useful results. (Bugh, 1984:10-12; Cviedo, 1980:147-148; Henry and others, 1981:144; Curtis, 1979:99-100)

Data metrics are measures based on the premise that software becomes more complex as the number of data item declarations and references increases (Hutchins, 1985:17).

Since the fundamental purpose of software is to process data (Conte and others, 1986:47), a complexity metric based on data items is intuitively appealing. Studies indicate that the amount and place of data use in a program has a definite effect on the effort required to develop the software (Oviedo, 1980:149). Since data flow analysis techniques are often used to measure the complexity generated by the data structures and their use, these metrics are often referred to as data flow metrics. Some of the better known of these are: a count of data item declarations, the span between data item references, and global data usage.

One of the simplest data metrics is a count of the amount of data used in a program. An easy way to obtain this count is from a cross-reference listing which shows all the variables declared and the lines where these variables are referenced. A compiler or assembler often has an option to generate this listing, and some software analyzers do, too. Computation of the metric is done by counting the times a declared data item is referenced. Care must be taken not to count unreferenced variables as well those that don't directly contribute to the solution of the problem. To be such a conceptually simple metric, data count can be quite involved to actually compute. (Conte and others, 1986:48-50)

Perhaps more important than the amount of data is the use of that data. Two aspects of this may be used to measure complexity: the span between data item references, and global data usage. Span is the "number of statements between

two textual references to the same identifier" (Elshoff, 1976:116). This is simply the number of statements that come between one occurrence of a variable and the next occurrence of the same variable. As the span of a data item increases, the complexity of the program is thought to increase since it is more difficult to follow the flow of the data item (Conte and others, 1986:56). Global data usage is common in today's modularized languages. A data item is often declared in one module and "shared" with other modules (this relates to the idea known as visibility in languages like Ada). A larger amount of sharing between modules is felt to indicate increased complexity because it is more difficult to find where a given data item is declared, used, or changed (Harrision and others, 1982:67). Many data metrics have been created but none are specifically discussed in this thesis since the actual metric is very language-dependent.

Information metrics are measures based on the premise that software becomes more complex as the flow of information (data items passed as parameters) between software system components increases (Henry and Kafura, 1981:511). This explains why they are also known as information flow metrics. Component complexity depends on two factors: the complexity of the individual component's code, and the complexity of the component's interface to other components (Henry and others, 1981:147). The best-known of these metrics was devised by Sallie Henry and Dennis Kafura.

Henry and Kafura's metric extends the data metric concepts. While data metrics concentrate on declarations and item usage, information metrics focus on the software component interface (i.e, the flow of information between components). This capability makes it possible to measure "complexity, module coupling, level interactions, and stress points" which is not possible with other metric categories such as size (Henry and Kafura, 1981:511). Several studies have indicated useful results with the information metric. (Rodríguez and Tsai; 1986:369-374; Henry and Kafura, 1981:511-517; Henry and others, 1981:145-149)

The final category of metrics represents a combination of the other categories. This approach is supported by researchers who feel that no single metric is capable of measuring software quality (Bowen, 1978:148; Boehm and others, 1980:220). Some of the reasons for this conclusion are:

- "good" quality is subjective--it varies with the user
- quality in one area is often gained at the expense of another area
- no metric yet devised fully measures what it is intended to measure

(Boehm and others, 1980:220)

A hybrid metric is often presented as a suite of orthogonal, non-overlapping measures, but it may also be a single metric which integrates the strengths of various metrics into one measure.

The Oviedo/VanVerth metric combines control and data measures into one. Originally conceived by Oviedo, the metric was extended by VanVerth to include procedure and function calls as well as between-module data flow. Some evidence of this measure's usefulness has been presented in the literature. (VanVerth, 1986:164,168)

Summary

This chapter began with a discussion of several attempts to define "software quality." A satisfactory definition was not found, but two dichotomous models which exhibited user-oriented characteristics at one end and measurement-oriented characteristics at the other were described. Different views of quality were provided by grouping the measurement-oriented characteristics in various ways. Although maintainability is the quality being studied, it was determined to be a quality that cannot be directly measured. However, complexity was identified as the biggest contributor of maintainability, and a number of complexity measures were presented.

Chapter 3 considers each of these metrics in greater detail, and picks the metrics which will be implemented.

III. Metric Selection

The literature review of Chapter 2 examined several approaches to quantifying software quality. Two models of software quality, five categories of complexity measures, and several complexity metrics described how it can be modeled and ultimately measured. Still, the measurement of software quality is not an exact science although studies indicate that some metrics can produce useful results. This thesis uses selected metrics to indicate software maintainability while recognizing that

> [metrics] is still a primitive technology and should be used by management and engineering as a tool to augment good judgement, not to replace it [Boehm and others, 1980:219].

The sponsor's present software review process is based on a software evaluation methodology (Peercy, 1981:343-351) which uses a closed-form questionnaire to determine software maintainability. A small evaluation team composed of at least five people who are knowledgeable about the software and maintainability answers a series of questions to provide

> enough specific information to identify for which parts of the software and for what reasons maintainability may be a problem [Peercy, 1981:343].

The questions prompt each evaluator on the team to quantify his subjective opinion about how well the software under review reflects desirable quality characteristics. Both documentation and source listings are reviewed. According to the methodology, at least 10 percent of all the modules in a JOVIAL (J73) program must be selected at random and reviewed

to insure the results are valid for the entire program. (Peercy, 1981:343-346; AFOTECP 800-2, Vol. 3, 28 Jan 88)

This thesis proposes that the sponsor's present manual review be augmented with selected metrics to provide a list of all the JOVIAL (J73) modules ordered from least to most maintainable. Because the manual review uses a relatively small sample of modules to indicate the maintainability of the entire program, most modules are never checked. A list indicating the relative position of all modules would permit some conclusions to be drawn about the unchecked modules based on the ones that were reviewed.

Criteria

The literature search indicates that certain metrics do measure maintainability (actually complexity), but they must be identified for implementation. However, each metric that is selected must meet criteria which support the problem, scope, and limitations of this study. The metric must:

- have empirical support,
- be applicable to JOVIAL (J73), and
- be automatable.

Empirical support for a metric was originally specified in the scope of this thesis effort. Of course, the metric must be applicable to JOVIAL (J73) or it would not be of any interest in this study. The last criteria was not included at the start of this effort, but became apparent through the literature search.

A metric which has empirical support is one that has been implemented and used in at least one significant experiment that concluded the metric did produce results which were consistent with the expected outcome. While this is by no means rigorous proof of the metric's validity, it is positive support of the metric's usefulness given the current state of software quality measurement. Although the kinds of systems studied and the quality of research varies so widely that strict comparison of results is nearly impossible, a lot of the research was done well and added significantly to the knowledge of metrics (Curtis, 1979:97). Since empirical support was a part of the scope, only metrics meeting it were reviewed in Chapter 2.

One might expect large software systems to be written in high-order languages (HOLs) to take advantage of the many well-documented benefits of HOLs over low-level languages. Since all of the metrics studies reviewed for this effort used HOLs (primarily FORTRAN but also some COBOL, Pascal, and even Ada!), this is apparently a good assumption. Many of the studies implied that a primary requirement of a metric is that it have general application to <u>any</u> programming language even though HOLs only were used. In any case, all of the metrics discussed in Chapter 2 may be applied to JOVIAL (J73) since it is a HOL.

Any metric that this study selects for implementation must be automatable since these metrics are superior to those

which must be computed manually. While all metrics may be manually computed, doing so may actually be self-defeating.

> In fact, if to evaluate a metric requires an expert to read a program and make a judgment, the numerical value will generally provide much less insight than the understanding that the expert will pick up in the evaluation process [Boehm and others, 1980:222].

Automated metrics alleviate tedious, error-prone work, remove the possibility of human bias in the computations, and are much faster than manual methods (Basili and Reiter, 1979:107-108). Once an analysis algorithm has been decided upon and implemented, the metric is computed in the same manner from then on. The people reviewing the software are free to concentrate on the quality of the software rather than the use of the software.

However, not all proposed metrics are automatable since some types of analysis are uniquely human in nature and can't be performed through programmed algorithms. A metric is automatable if:

> "(1) The data can be collected without interfering with individuals involved in the development,
>
> (2) the measures are computed algorithmically from quantifiable sources normally available, and
>
> (3) the measures can be reproduced on other projects by essentially the same algorithms" (Basili and Reiter, 1979:107).

Because of the way this study will use the selected metrics, all of these requirements are easily met. First, the metrics will be computed from released (or product) code. No interference with the developers is possible since they have already finished their work. Second, source code

will always be available since the sponsor's method uses it as well. Finally (and most importantly), the metrics can be used directly on any set of JOVIAL (J73) modules. Especially of value is the ability to compare metrics computed on future versions of the same program.

The use of automated metrics to measure software quality was considered in the software evaluation methodology used in the sponsor's manual review but was rejected

> because of the number of software systems to be evaluated, the variability (language, computer, functions) of the software to be evaluated, and the limited state of the art in practical automated evaluation tools [Peercy, 1981:343].

This study recoginizes these limitations but overcomes them by applying selected metrics under certain situations. A general assessment of software quality is not required--only the maintainability of JOVIAL (J73) modules. These results may then be used to rank all of the modules from least to most maintainable. The sponsor can then use this list to decide the probable maintainability of modules that the evaluation team did not review.

Comparison

The possible implementation of each metric introduced in the previous chapter will be discussed. The primary interest in this section is whether the metric meets the selection criteria explained above.

The lines of code (LOC) metric is the oldest and most used measure of software size. It holds this distinction

because it is a simple measure that is relatively easy to compute, and it applies quite well to programming languages which allow only one line of code to a physical line in a source program (for example, FORTRAN, BASIC, or assembler). Size metrics like LOC are based on the assumption that a larger program is more complex so the lines of source code may simply be counted to produce the metric (Harrison and others, 1982:66). Unfortunately, actual computation has never been that simple since disagreement over what makes up a "line of code" has always persisted. The argument has traditionally centered on whether data declaration and comment lines should be counted along with executable source lines of code. More important to this study is a relatively new point of disagreement: counting logical lines of code. Used extensively in modern HOLs, logical lines extend over several physical lines, and computation of the LOC metric is more difficult because some counting algorithm must be used. Although various solutions to this problem may be found, no single, clear-cut answer is available. Despite the empirical support for the LOC metric, it does not appear useful to this study due to computational difficulties caused by the lack of a definite algorithm for handling the free-form coding style permitted in JOVIAL (J73) (a physical line of code does not equate to a logical line of code). (Levitin, 1986:314-315)

The statement count metric evolved because the LOC metric proved to be less applicable to modern HOLs. Though not a definition, a statement may be considered a single

logical line of source code. Although the perspective is different, the statement count metric has definition problems very similar to those of the LOC metric. Most modern HOLs permit the use of structured statements such as compound statements (delimited by BEGIN-END), repetitive statements (WHILE, REPEAT, and FOR), and conditional statements (IF and CASE). All contain embedded statements, and this greatly confuses the metric computation since some algorithm for counting these structured statements must be used. While most languages are strongly statement-oriented, some don't even support the concept of a statement (for example, APL, LISP, and PROLOG). Terminology is also a problem since a statement may be known by different names in different languages. These problems make the statement count metric undesirable for use in this study. (Levitin, 1986:315)

The token count metric is a further refinement of the two metrics discussed above. While the LOC metric measures a physical line of code and the statement count metric measures a logical line of code, the token count metric measures the tokens which every line (physical or logical) of source code can be broken into. A token is

> the basic syntactic unit from which a program can be constructed. Each token represents a sequence of characters that can be treated as a single logical entity. At the same time, these entities are atomic, i.e., they have no possible further subdivisions [Levitin, 1986:316].

As an example, consider the following program fragment.

```
AA = BB ** 2;
```

This line of code contains six tokens. Two tokens represent variable names (AA and BB), two tokens represent operators (= and **), one token represents a numeric literal (2), and one token represents a line of code terminator (;). An important consideration of the token count metric is that tokens are always the first product of the compilation process so the metric may be obtained for any language. One significant disadvantage is that a large program will contain an enormous number of tokens. This should not be considered a serious detriment since the numerical values of many metrics can be quite large. The token count metric can be easily automated for JOVIAL (J73), and appears useful for this study. (Levitin, 1986:315-316)

Software Science metrics are among the most widely known and used software measures (Harrison and others, 1982:65). Two of them, length and volume, are of interest in this study. All of the Software Science metrics are based on counts of so-called operators and operands. Generally, an operand may be considered to be any variable or constant and an operator to be any symbol affecting the operands (Lecciso and others, 1986:605-606). The metrics are defined as:

Length $N = N_1 + N_2$

Volume $V = N * \log_2 (n_1 + n_2)$

where n_1 is the number of unique operators
n_2 is the number of unique operands
N_1 is the total number of operator occurrences
N_2 is the total number of operand occurrences
(Lecisso and others, 1986:606; Levitin, 1986:316)

Looking at these definitions, it can be seen that length is simply a total of the operands and operators in a program. This is very similar to the previously discussed token count. Volume is more complex since it involves a base-2 logarithm. This metric attempts to provide a measure of the bits needed to contain the program in the computer's memory. Both of these measures suffer from unsupported assumptions. First, no clear, unambiguous definition of operators and operands exists. This alone casts a questionable shadow on the value of any of the Software Science metrics since a cornerstone of research is repeatability. Second, no logical reason exists for the equations of some of the metrics, such as volume. Even though the length and volume metrics have impressive empirical support, these criticisms appear serious enough to avoid using either metric in this study. (Lecciso, 1986:605-608)

The Cyclomatic Complexity Number, V(G), is based on the assumption that complexity increases as the number of control flow paths through a program increase. Of course, loop instructions make a simple count of control paths impractical since the number could be infinite. To avoid this, the Cyclomatic Complexity Number is defined as the "maximum number of linearly independent circuits in the flow graph" (Bugh, 1984:10). "Simply stated, [this] metric counts the number of basic control path segments in a computer program" (Curtis, 1979:99). Since each path represents an important link in understanding the logic of the program, this metric

provides an intuitive measure of complexity (Henry and others, 1981:144). In mathematical form, the computation is

V(G) = # of edges - # of nodes + 2p

where an edge represents a branch in the program control logic

a node represents a section of sequentially executed statements

p represents the number of connected components in the flow graph

(Curtis, 1979:99)

The Cyclomatic Complexity Number has been criticized as being unable to adequately measure the complexity of nested control structures. Consider the following two fragments.

```
if P1 then                    if P1 then
   if P2 then                    S1
      S1                      else S2
   else S2                    if P2 then
else S3                          S3
```

Each structure performs a different task yet both have the same Cyclomatic Complexity Number. More importantly, the one on the right seems simpler to understand. However, empirical results of this metric have been good, and its theoretical basis is sound, so the Cyclomatic Complexity Number appears useful to this study. (Lecciso and others, 1986:608)

Data metrics assume that a more complex program will contain more data item declarations and references to those items (Hutchins, 1985:17). Three of these metrics were discussed in Chapter 2: a count of data item declarations, the span between data item references, and global data usage. All may be considered important measures of complexity since

> the execution of a computer program normally implies input of data, operations on [the data], and output of the results of these opeations in a sequence determined by the program and the data [Fosdick and Osterweil, 1976:306].

Data count metrics involve a count of the data items input, output, or declared in a program. A more complex program is expected to have more data items. However, a simple count of data items is not sufficient to compute the metric. Care must be taken to include only items which are actually referenced. In addition, some data items which don't contribute to the solution of the task must also be excluded from the count. Although computation of this metric appears to be algorithmic, it is actually subjective (Conte and others, 1986:50). This, combined with a lack of useful empirical evidence, make this metric undesirable for use in this study. (Conte and others, 1986:48-50)

Data span metrics measure the number of statements that occur between two successive references to the same data item (Conte and others, 1986:56). They are "based on the locality of data references within the program" (Harrison and others, 1982:67). Although this metric has been used in at least one study (Elshoff, 1976:116), little empirical evidence exists to support it. Furthermore, data span suffers from the same definition problem as the statement count metric since both measures depend on statements. If a statement cannot be adequately defined, the measure is probably not repeatable. Because of these problems, the data span metric is not suitable for use in this study.

Global data usage metrics measure the amount of data that is declared in one component and referenced in others. This is especially important since modern HOLs usually allow data item declarations to be made in separate modules and globally referenced. Program complexity is felt to increase as the number of "shared" data items increases (Conte and others, 1986:57). The theoretical basis of this metric appears valid and data flow analysis techniques are easily automated. The global data usage metric meets all of the selection criteria and appears to be useful to this study. (Conte and others, 1986:47-59)

The information metric extends the techniques of data flow analysis to deal

> directly with the system connectivity by observing the flow of information or control among system components [Henry and Kafura, 1981:511].

It assumes that the complexity of a segment of source code depends on two factors: the code within the segment, and the segment's environmental connections.

This approach has some advantages. The metric reflects connectivity between segments better than other metrics since passed parameters and global data structures are considered. It indicates "module coupling, level interactions, and stress points" because the data items passed in and out of a segment are measured (Henry and Kafura, 1981:511). Some metrics concentrate on a single characteristic of complexity, but the information flow metric uses both size and data since segment length and data flow are measured. Finally, data flow

techniques are fairly standard and can be automated without much difficulty. The information metric is defined as

$$\text{length} * (\text{fan-in} * \text{fan-out})^2$$

Length refers to a measure of the size of the module, and indicates the module complexity component of the metric. To compute length, the originators of the information metric used LOC for convenience, but noted that other metrics such as the Cyclomatic Complexity Number had stronger empirical evidence. The token count metric is preferred in this study for convenience as well as the reasons discussed earlier in this section. Fan-in is the number of parameters passed into a module plus the number of data items the module accesses globally. Fan-out is the number of parameters passed from a module plus the number of data items the module updates globally.

> The term fan-in * fan-out represents the total possible number of combinations of an input source to an output destination. The weighting of the fan-in and fan-out component is based on the belief that the complexity is more than linear in terms of the connections which a procedure has to its environment [Henry and Kafura, 1981:513].

The power of two was selected as the weighting factor because other studies had used it to relate human interaction with software. This metric appears to have a valid theoretical basis and is easily automatable. It seems to be a strong indicator of complexity and useful to this study. (Henry and Kafura, 1981:510-517)

Hybrid metrics are measures of software quality composed of either a suite of metrics like the ones already discussed

or a single metric that incorporates more than one underlying theoretical measurement concept into one approach. While a suite of metrics sounds appealing, several problems exist. First, the individual metrics must be independent of one another to avoid a simple duplication of information. Some studies have been done to determine independence, but the results are not totally conclusive since the applications and usages are so varied. Second, guidance for interpreting the results is not readily available. An improvement of one software quality is often gained at the expense of another (Boehm and others, 1980:220). Given a group of metrics that indicate one quality is low, another is high, and so on, how does one make an overall evaluation? The combined metric may be a more plausible approach if the theoretical basis can be shown (at least empirically). However, very few studies to do this have been conducted. While hybrid metrics are intuitively appealing, the problems associated with their interpretation and theoretical foundations make them unuseful for this study.

Selection

The preceding section discussed each metric studied in this thesis to identify the ones that appeared to be the most beneficial to implement. Particular emphasis was placed on those metrics with solid empirical evidence as well as on the other two criteria, applicability to JOVIAL (J73) software and automatability. Although each metric has been given

equal treatment up to this point, the sponsor had initially expressed a desire to de-emphasize size and control metrics in the selection process. Accordingly, emphasis will be shifted away from size metrics even though these are the oldest, most studied metrics and,

> in general, metrics based on measures of program size have been the most successful to date, with experimental evidence indicating that larger programs have greater maintenance costs than smaller ones [Li and Cheung, 1987:707].

However, the sponsor felt that

> [size metrics are] too simplistic since the length of a piece of code is not a complete indicator of its complexity. The tasks performed, the control structures used, and the program variables referenced also contribute to complexity [McClure, 1978:150].

Likewise excluded from true consideration, control measures have been shown to produce useful results.

> Evidence continues to mount that metrics developed from graphs of the control flow are related to important criteria such as the number of errors existing in a segment of code and the time to find and repair such errors [Curtis, 1981:210].

But some studies have failed to show the usefulness of the control metrics. Taking a fresh look at some of the newer metrics could identify ones that might measure software maintainability even better than size and control metrics. In the interest of completeness, however, all five of the metrics categories presented in Chapter 2 were reviewed.

The comparison of the metrics introduced in Chapter 2 identified several metrics which may be implemented in this study. Included are:

- token count metric
- Cyclomatic Complexity Number
- global data usage metric
- information metric

Based on the sponsor's guidance, token count and Cyclomatic Complexity Number will be removed from further consideration for implementation in this thesis. This leaves these metrics as this study's candidates for implementation:

- global data usage metric
- information metric

Implementation of both of these candidate metrics is not possible in this study due to time constraints. To make the choice, consideration must be given to the problem to be solved, the resources needed by the solutions, and the theoretical basis of the approaches. The goal of the metric selected is to indicate the degree to which a JOVIAL (J73) program possesses the software quality of maintainability. To do this, the complexity of the program will be measured because maintainability cannot be measured directly. Since each candidate is a complexity metric, each is capable of solving the problem.

The global data usage metric can only be computed if a complete view of all the data items in the program is available. Local data declarations are easy to handle but global data declarations greatly complicate computation of the metric. Although good JOVIAL (J73) programming technique requires that global items should only be referenced through

a particular mechanism, the language does not require it. This means that an accurate measure can be made only if the entire program is analyzed at once. This is a giant task for a large JOVIAL (J73) program since it may be composed of many segments.

On the other hand, the information metric may be computed on individual segments (Henry and Kafura, 1981:512). This alleviates the need to analyze the whole program at one time, and greatly reduces the amount of work required to implement the metric.

While the selection criteria is equally fulfilled by each metric, the information metric may have a slightly better measurement capability because it uses two views of complexity rather than just one. The global data usage metric employs a data view of complexity since it measures the amount of data that is declared in one segment and used in another. The information metric employs both a data view coupled with a size view of complexity since it measures the amount of data flowing in and out of a segment but factors in the length of the segment.

Based on its less demanding implementation and possible extra capability, the information flow metric is selected for implementation in this study.

Chapter 4 describes the implementation details that had to be addressed to program the information metric for JOVIAL (J73) in the Ada language.

IV. Implementation of Selected Metrics

In the previous chapter, the metrics studied in this thesis were compared, and the information metric was selected for implementation. This chapter describes the steps taken to implement that metric, but before getting into the details, a brief overview of the JOVIAL (J73) language is instructive.

Language Overview

JOVIAL (J73) is a high-order language (HOL) which is currently accepted by the Air Force as a standard programming language. It is a powerful, full-featured language that provides these capabilities:

- modular construction of programs
- block-structured control statements (loops, IF, and CASE)
- a restricted GOTO statement
- strong type checking to enforce data abstractions
- low-level control of machine-specific operations and storage
- direct addressing of storage units
- assignment of implementation-specific machine parameters to aid portability

(SofTech, 1988:1-7)

Several other features of special interest are: free-form coding style, pointers, simple and composite data structures, and direct linkage to code written in other programming languages. The ability to easily link to other languages is vital because JOVIAL (J73) contains no means to directly input and output data. (FORTRAN subroutines are

frequently used to perform input/output operations.) The language is used primarily for flight control, command and control, and avionics applications where input/output requirements are minimal. (SofTech, 1984:1)

As Figure 6 shows, a JOVIAL (J73) program consists of only one main program module and none or more procedure or compool modules. These modules are separately compiled and linked together at a later time to form a complete program. Each module contains statements, subroutines, or a mixture of both, and is delimited by the reserved words START (at the beginning of the module) and TERM (at the end of the module). (SofTech, 1984:241-252)

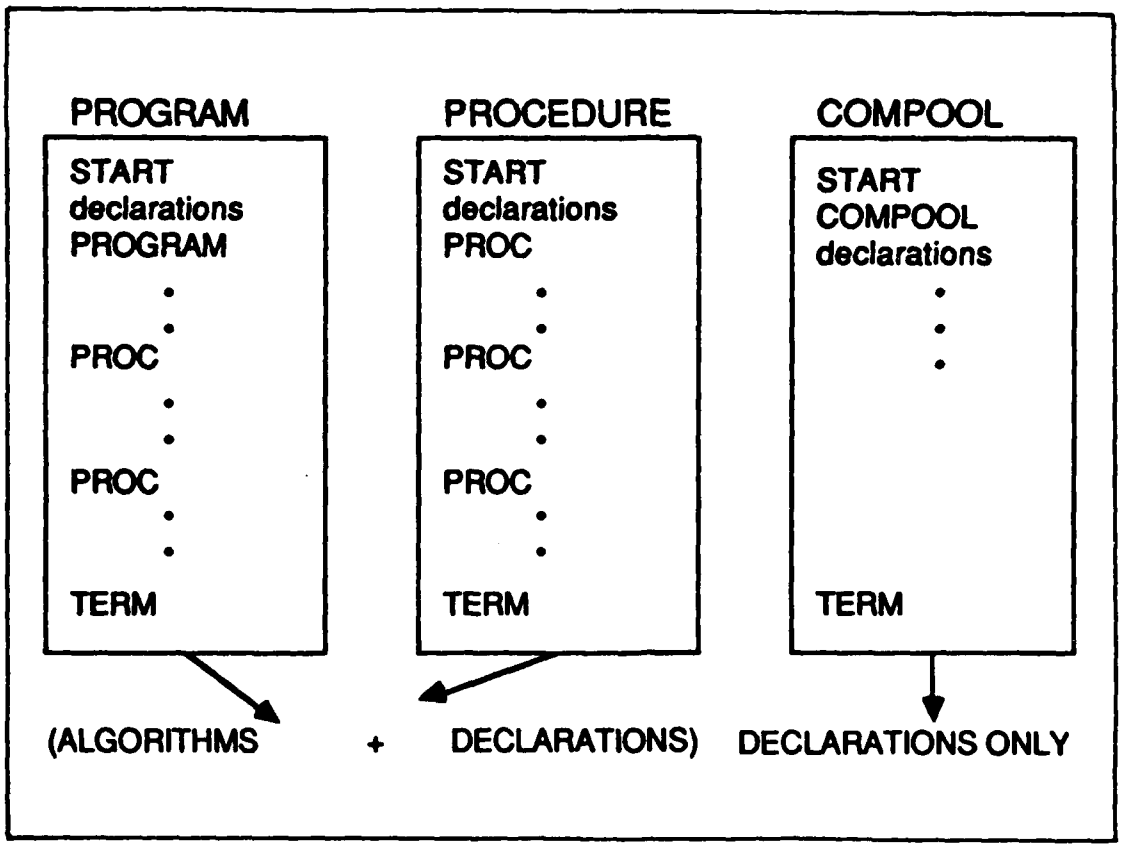

Figure 6. Three Kinds of JOVIAL (J73) Modules (SofTech, 1988:1-24)

Program execution begins in the main program module with a special subroutine which starts with the reserved word PROGRAM followed by the name of the complete program. Unlike ordinary subroutines (which begin with the reserved word PROC), no parameters may be passed into the PROGRAM subroutine. Execution continues until the last statement in the PROGRAM subroutine is reached, unless a termination statement is found first. (SofTech, 1988:9-16 thru 9-21)

The variety of statements supported by JOVIAL (J73) is quite diverse. Both simple and compound forms of these constructs are permitted: simple and multiple assignments, IF-ELSE, CASE, WHILE and FOR loops, and subroutine calls (which may be either procedures or functions). (SofTech, 1984:183-208)

JOVIAL (J73) program structure is as rich as its constructs. Several data structures are allowed: simple data items, tables of simple data items, and blocks of data declarations composed of simple data items and tables, as well as nested blocks. A complete suite of data types including signed and unsigned integers, fixed point numbers, floating point numbers, enumeration values, character strings, bit strings, and pointers are supported. (SofTech, 1984:53-108)

Adaptation Attempts

In an attempt to rapidly implement the information metric, an effort to adapt either the JOVIAL Automated Verification System (JAVS) or the Integrated Tool Set (ITS)

JOVIAL (J73) compiler was made. Since both these software packages perform an in-depth analysis of JOVIAL (J73) code, all the data needed to compute the metric would be readily available. Unfortunately, several months passed before a usable copy of JAVS could be obtained. This delay caused it to be dropped from serious adaptation intentions. The ITS compiler was much easier to get. After a preliminary review of the source code, however, it became clear that the compiler was a very complicated software package. It was decided that enough time was not available to gain the necessary familiarity needed to adapt the compiler to compute the information metric.

A discussion with the contractor who maintains the ITS compiler confirmed the wisdom of this decision. The compiler has been modified many times since its initial release. It was adapted to five host computers and configured for about 25 target computers. As the language matured, the compiler became a mixture of old and new language features. In short, modifications such as adding the capability to compute the information metric could realistically be made only by personnel who possess a lot of experience with ITS compiler. This made the final option, writing new code, the only alternative for this study. (Engimann, 1988)

Construction Analysis

With the decision to write new code made, the next step was to determine exactly how to construct an analyzer to

compute the information metric. Since the sponsor had requested that any new code be written in the JANUS/Ada language, the implementation language was already selected. To determine a suitable internal representation for the analyzer, three questions had to be answered.

First, which features of JOVIAL (J73) affect the computation of the information metric?

A language as rich as this has many features that must be considered. The ones that will be addressed here include:

- program structure
- type declarations
- data declarations
- formulae
- built-in functions
- conversion
- statements
- subroutines
- externals
- directives
- define capability
- advanced topics

The information metric is defined in terms of the flow of "effective parameters" through a segment of code (Henry and others, 1981:145-147). A segment refers to any one of a number of separate groups of code which form a single program when taken together. Originally, this metric was validated using the UNIX operating system (Version 6) so a procedure

was assumed to be a segment (Henry and Kafura, 1981:513). This was a logical choice since a program written in C (the UNIX implementation language) consists of several independent procedures that call and are called by one another (Waite and others, 1987:278). A similar choice must be made for this study. A JOVIAL (J73) structure that equates to a segment must be identified. A JOVIAL (J73) program may be viewed as a number of subroutines partitioned into modules to allow separate compilation. This naturally suggests that a segment is equivalent to a subroutine in this language. Even though the metric is referenced to subroutines, emphasis remains on both the passed parameters and shared global data structures; a simple subroutine-by-subroutine analysis provides the data needed to determine the information metric (Henry and Kafura, 1981:513).

Because a metric will be computed for each subroutine that occurs within a module, an overall assessment of module complexity is possible (Henry and Kafura, 1981:514). Main program and procedure module complexity is a combination of the individual subroutine complexities. Compool modules, on the other hand, have an information metric of zero. The purpose of a compool module is to help control shared global data. A data item or subroutine that is externally declared (DEF'd) becomes a shared global item since it can be referenced (REF'd) by other modules (SofTech, 1984:236). Subroutines may be declared in either the main program module or a procedure module, but they may not be declared in a

compool module. It may contain only declarations: constant, type, define, overlay, DEF specifications for data or statements, and REF specifications for data or subroutines (SofTech, 1984:243-248). Thus, a compool module has no information flow since subroutines cannot be declared.

Type statements simply establish a data class and do not contribute to the information metric except through the length component.

Data declarations are important to the computation of the information metric for two reasons. First, subroutine formal parameter lists require a declaration for each entry. If a procedure were defined like this

```
PROC CUBE'ROOT ( NUMBER : ROOT );
```

then the variables NUMBER and ROOT would have to be declared in the body of the procedure. Second, declared variables that do not appear in the formal parameter list must be local declarations and should not be considered information flow.

In JOVIAL (J73), a formula "describes the computation of a value" (SofTech, 1984:117). Formulas are important to the computation of the information metric because global data structures may be referenced. If a global appears on the right side of an assignment operator, it is fan-in to the subroutine; if on the left, it is fan-out of the subroutine.

JOVIAL (J73) provides some predefined capabilities that give the programmer a way of getting information that would otherwise be inaccessible or difficult to calculate (SofTech, 1984:137). Built-in functions are invoked just like other

functions but do not have to be defined. Typically, these subroutines offer low-level data manipulation capabilities. Because of this, this study will not consider the returned values as fan-in. References to global data structures in the function argument, however, will be considered fan-in.

Variables are strongly typed in JOVIAL (J73), but ways to convert from one type to another are provided. Sometimes, the compiler does the conversion automatically, but other times an explicit conversion is required (SofTech, 1984:161). As with type statements, explicit conversion operators do not contribute to information flow.

Statements in JOVIAL (J73) may be either simple or compound. Simple statements "perform computations, control program flow, and call [subroutines]" (SofTech, 1984:183). Included are the following:

- assignment
- IF
- CASE
- LOOP
- EXIT
- RETURN
- ABORT
- STOP
- GOTO
- subroutine call

(SofTech, 1984:183)

Compound statements are groups of simple statements delimited by the reserved words BEGIN and END. Assignment statements were previously discussed in conjunction with formulas. IF, CASE, and LOOP statements may use global data structures (as predicates) but cannot update them. Thus, only fan-in may occur in one of these statements. A single letter variable name may be used in the LOOP statement. It does not have to be declared and has no meaning outside of the loop (SofTech, 1984:203). These single letter variables should not be considered as information flow since they are local declarations. EXIT, RETURN, ABORT, and STOP are reserved words and make no contribution to information flow. The GOTO statement is part of program control flow only; it provides no information. The associated labels (which may precede any statement) must be ignored. Subroutine calls contribute directly to information flow. If the subroutine is a procedure, the parameters (actual list) indicate both fan-in and fan-out. If it is a function, the subroutine name indicates fan-in while the parameters indicate fan-out (and fan-in, if any).

A subroutine definition describes "a self-contained portion of a program [that] interacts with its environment through its parameters or global data" (SofTech, 1984:209). The definition begins with the reserved word PROC followed by the subroutine name, use attribute (recursive or reentrant), parameters (formal list), and the subroutine body which may be either simple or compound. A simple body consists of only

one statement while a compound body contains many statements delimited by the reserved words BEGIN and END. It should be obvious that subroutine definitions are the main structures of concern in the computation of the metric. Even the main code (begun with the reserved word PROGRAM) is a subroutine.

As described in the language overview at the beginning of this chapter, a JOVIAL (J73) program may consist of three kinds of modules: one main program module and none or more procedure or compool modules. A module may be a mixture of statements and subroutine definitions, and is provided to support separate compilation (SofTech, 1984:235). Compiled modules share parameters and data structures through external names (SofTech, 1984:235). Names are declared external with a DEF specification, and referenced with a REF specification. Although DEF-REF relationships indicate global information, it would be difficult to compute the information metric if all these relationships in a program had to be found. To do this, the entire program would have to be analyzed at once. This approach has already been discussed and rejected in Chapter 3. DEF-REF relationships will not used to compute the metric.

Directives are reserved words that provide additional information to the compiler (SofTech, 1984:255). As such, they do not contribute to information flow.

JOVIAL (J73) provides a define capability which may be used to "associate a name with a string...of text" (SofTech, 1984:279). Anyplace the name appears in a program, it is

replaced by the compiler with the string of text. Although the defined name is not information flow (but may be treated as a local declaration), the string could be information flow if it contains references to global data structures.

Several advanced features are available in JOVIAL (J73). Generally, they give the programmer the ability to work with memory directly and communicate with low-level devices. While these capabilities are useful in programs, they are of little consequence to information flow.

Second, what data is needed to compute the information metric?

The length, fan-in, and fan-out is needed to compute the information metric for a subroutine. The metric formula incorporates two factors of complexity: code and parameter (Henry and Kafura, 1981:513). Length provides a measure of code complexity. Although LOC was used in the original work (Henry and Kafura, 1981:513), a token count will be used by this study for reasons discussed in Chapter 3. The fan-in/ fan-out term of the formula provides a measure of parameter complexity. The fan-in and fan-out of a subroutine will be obtained from the definition parameters (formal list), the call parameters (actual list), and the global data structures which are referenced.

The formal parameter list separates fan-in and fan-out names with a colon. If no colon is included, all the names are assumed to be fan-in. As this subroutine definition shows:

```
PROC PICK'MIN ( AN'ARRAY : MIN'NUM );
```

AN'ARRAY is the only fan-in name. The other name, A'NUM, is a fan-out name. Now consider this definition:

```
PROC PICK'MIN ( AN'ARRAY );
```

In this case, only one parameter is shown, and it is fan-in. The next definition illustrates the final case:

```
PROC PICK'MIN ( : MIN'NUM );
```

Again, only one parameter is shown but it is fan-out since a colon precedes it. Note that it could be both fan-in as well as fan-out but this cannot be determined since only syntactic analysis is used. (SofTech, 1984:212, 217-222)

The actual list (used in a subroutine call) also uses a colon as a separator. The parameters are arranged in the order expected by the subroutine definition. The analyzer must, therefore, reverse the fan-in and fan-out calculations of the subroutine. This is illustrated in Figure 7.

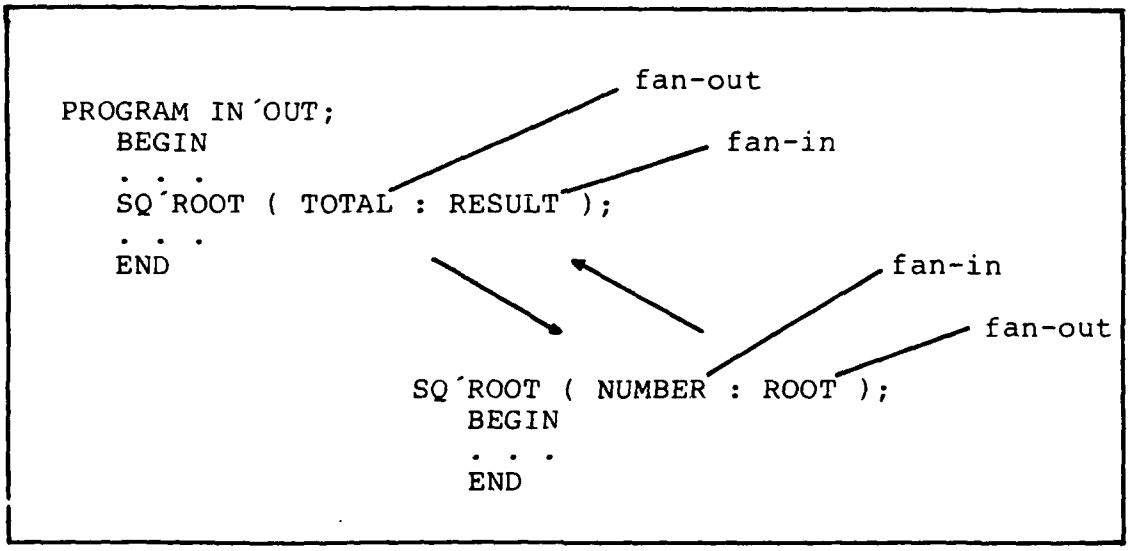

Figure 7. Fan-in/Fan-out Reversal

Static analysis cannot determine whether a parameter right of the colon in the actual list of a subroutine call is only fan-in or both fan-in and fan-out. To compensate for this, these parameters will be considered as both fan-in and fan-out in every case. This is a conservative solution since the risk of missing a flow of information is completely eliminated. The information metric should, therefore, be viewed as a conservative measure of complexity.

Simple syntax analysis can indicate whether a call is a procedure or function call. All of the preceding subroutine call examples are procedure calls since the returned value is not assigned. An equivalent function call for the second example is:

```
MIN'NUM = PICK'MIN ( AN'ARRAY );
```

The value returned by the subroutine call is assigned to MIN'NUM. This is a function call since, by definition, the returned value must be assigned. (SofTech, 1984:215)

Since cross-referencing between modules will not be done, names which are not declared locally will be considered global data references. Consider this definition:

```
PROC PICK'MIN ( A'NUM : MIN'NUM );
   BEGIN
   ITEM A'NUM U;              % declarations %
   ITEM MIN'NUM U;
   ITEM TEMP'NUM U;
   IF A'NUM < SUM;            % executable statements %
      TEMP'NUM = A'NUM;
   ELSE
      TEMP'NUM = SUM;
   MIN'NUM = TEMP'NUM;
   END
```

The declarations for A'NUM and MIN'NUM are required because they appear in the formal list. TEMP'NUM is also declared but it is a local declaration since it was not in the formal list. Any other name used in the executable statements will be interpreted as a reference to a global data structure. In this example, SUM will be taken as a global data item.

Third, how can the analyzer be best structured for computing the information metric?

Although any method of program development can be used with Ada, object oriented design (OOD) is often selected since the language provides many features which support OOD concepts. To construct the analyzer using OOD, these steps were followed:

"- Identify the objects and their attributes.

- Identify the operations that affect each object and the operations that each object must initiate.

- Establish the visibility of each object in relation to other objects.

- Establish the interface of each object.

- Implement each object." (Booch, 1987:48)

Rigorous application of OOD was not necessary because the design of the analyzer is based largely on compiler design theory which uses a well-established structure. A compiler performs two primary functions: analysis of the source program, and synthesis of a corresponding machine-language program. To do this, it is centered around a parser which performs syntax-directed analysis of a source program. A scanner provides the parser individual tokens for analysis.

After forming a structure based on the order in which the tokens occurred, the parser invokes semantic routines to determine the meaning of the structure. A code generator finishes the analysis by producing machine-language code. A complete compiler is illustrated in Figure 8.

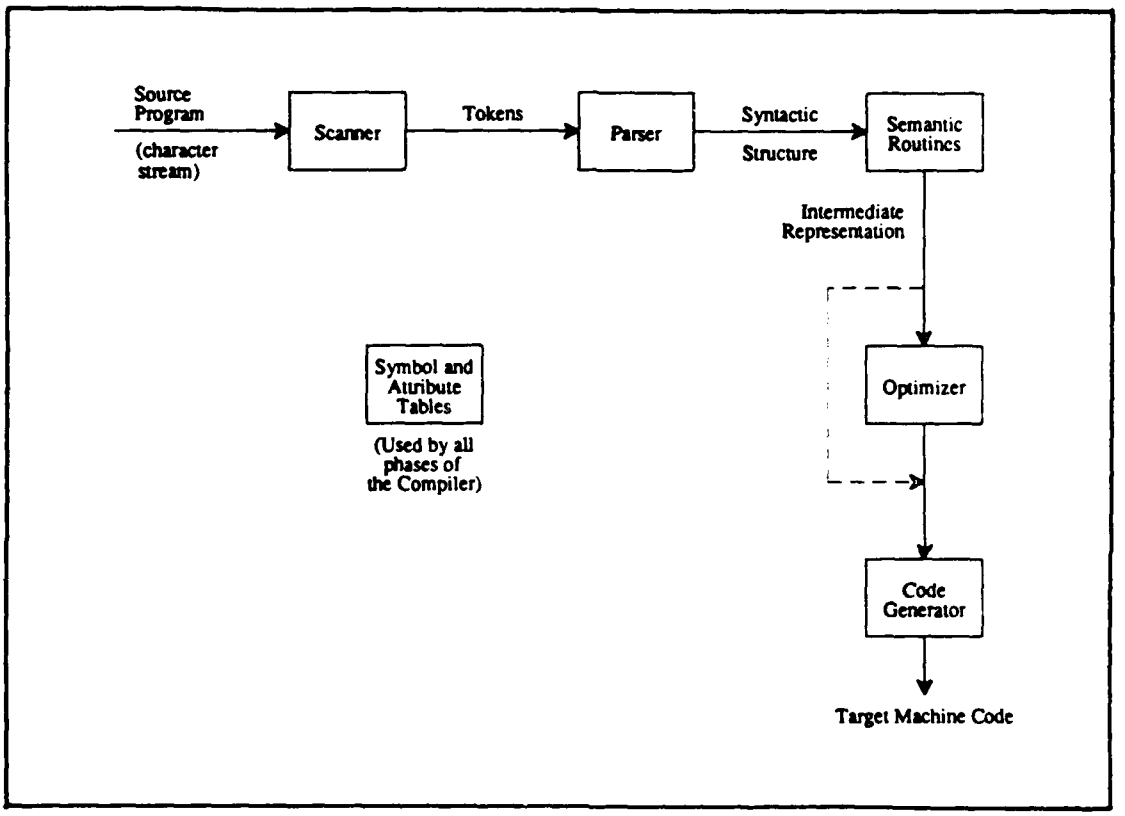

Figure 8. The Structure of a Syntax-Directed Compiler (Fischer and LeBlanc, 1988:9)

To compute the information metric, JAMS needs only the compiler front end. A scanner to provide the tokens and a parser to determine syntactic structure are enough for this application. In fact, the parser may even be a simple one since semantic routines are not required. In OOD terms, two objects are already known (the scanner and the parser) based on compiler theory. The rest of the design was driven by OOD

techniques such as data hiding, abstraction, coupling, and cohesion. (Fischer and LeBlanc, 1988:2-21)

This study uses recursive descent parsing, one of the simplest parsing techniques known. As each token is analyzed, a sequence may be recognized and a corresponding parsing procedure invoked (Fischer and LeBlanc, 1988:36-37). The advantage of this technique is that it is very simple to apply. The disadvantage is that each parsing procedure must be coded by hand. This is inefficient and time consuming since changes are difficult to make. While this disadvantage would be severe for a full compiler, it is not serious for this application because the semantic routines are not used and do not have to be written. Only syntax-directed analysis is needed. (Fischer and LeBlanc, 1988:115,197-200)

The answers to these three questions provide the basis for rules to construct an analyzer which will compute the information metric for JOVIAL (J73) software. Specific rules will make analyzer development more orderly and lessen the risk of overlooking language features. Table 3 lists these rules and introduces Figure 9.

Analyzer Construction

Although this design is not a strict OOD implementation, it does emphasize the objects and their associated operations required to analyze a JOVIAL (J73) program. Figure 10 shows the analyzer structure produced by this methodology (Booch, 1987:47-59). JAMS is the analyzer's abbreviated name which

Table 3

Rules for JAMS Construction

Rule 1.	An information metric will be computed for each subroutine.
Rule 2.	Since the information metric for a compool module will always be zero, a token count will be provided as a size metric.
Rule 3.	Computation of the information metric will be based on the number of tokens within a subroutine and its "effective parameters" (Henry and others, 1981:145-147) which are the passed parameters plus the global data items that are referenced.
Rule 4.	Type statements will not contribute to information flow.
Rule 5.	Local declarations will not contribute to information flow.
Rule 6.	Reference to a global data structure in a formula will be analyzed as follows: if on the right of an assignment operator, it is fan-in; if on the left, it is fan-out
Rule 7.	Built-in function return values will not contribute to information flow.
Rule 8.	Reference to a global data structure in a built-in function argument will be analyzed as fan-in.
Rule 9.	Explicit conversion operators will not contribute to information flow.
Rule 10.	Reference to a global data structure in a IF, CASE, or LOOP statement will be analyzed as fan-in.
Rule 11.	Single letter LOOP variables will be treated as local declarations.
Rule 12.	GOTO statements and their associated labels do not contribute to information flow.
Rule 13.	The main program module PROGRAM subroutine will be analyzed as a subroutine with no parameters.

- continued -

Table 3 - continued

Rules for JAMS Construction

Rule 14. An equal number of BEGIN and END statements will indicate complete analysis of a subroutine.

Rule 15. Program modules will be analyzed separately.

Rule 16. DEFINE'd names will be considered as local declarations. The associated string will be analyzed for global references.

Rule 17. Parameters in the formal list of a subroutine definition will be analyzed as follows: names to the left of the colon are fan-in, names to the right are fan-out.

Rule 18. Parameters in the actual list of a subroutine call will be analyzed as follows: names to the left of the colon are fan-out, names to the right are both fan-in and fan-out.

Rule 19. If the subroutine is a function, the subroutine name given in the call will be considered fan-in.

Rule 20. Module structure will always be assumed to be as shown in Figure 9.

Main-program-module Structure

```
START
COMPOOL declarations
. . .
declarations and REF's
. . .
PROGRAM program-name;
declarations
. . .
   BEGIN
   body of program (statements and subroutines)
   . . .
   local subroutines
   . . .
   END
subroutine DEF's
. . .
TERM
```

Procedure-module Structure

```
START
COMPOOL declarations
. . .
declarations and REF's
. . .
subroutine DEF's
. . .
TERM
```

Compool-module Structure

```
START
COMPOOL declarations
. . .
COMPOOL compool-name;
   BEGIN
   declarations and REF's
   . . .
   END
TERM
```

Figure 9. JOVIAL (J73) Module Structure (SofTech, 1988:9-17 thru 9-37)

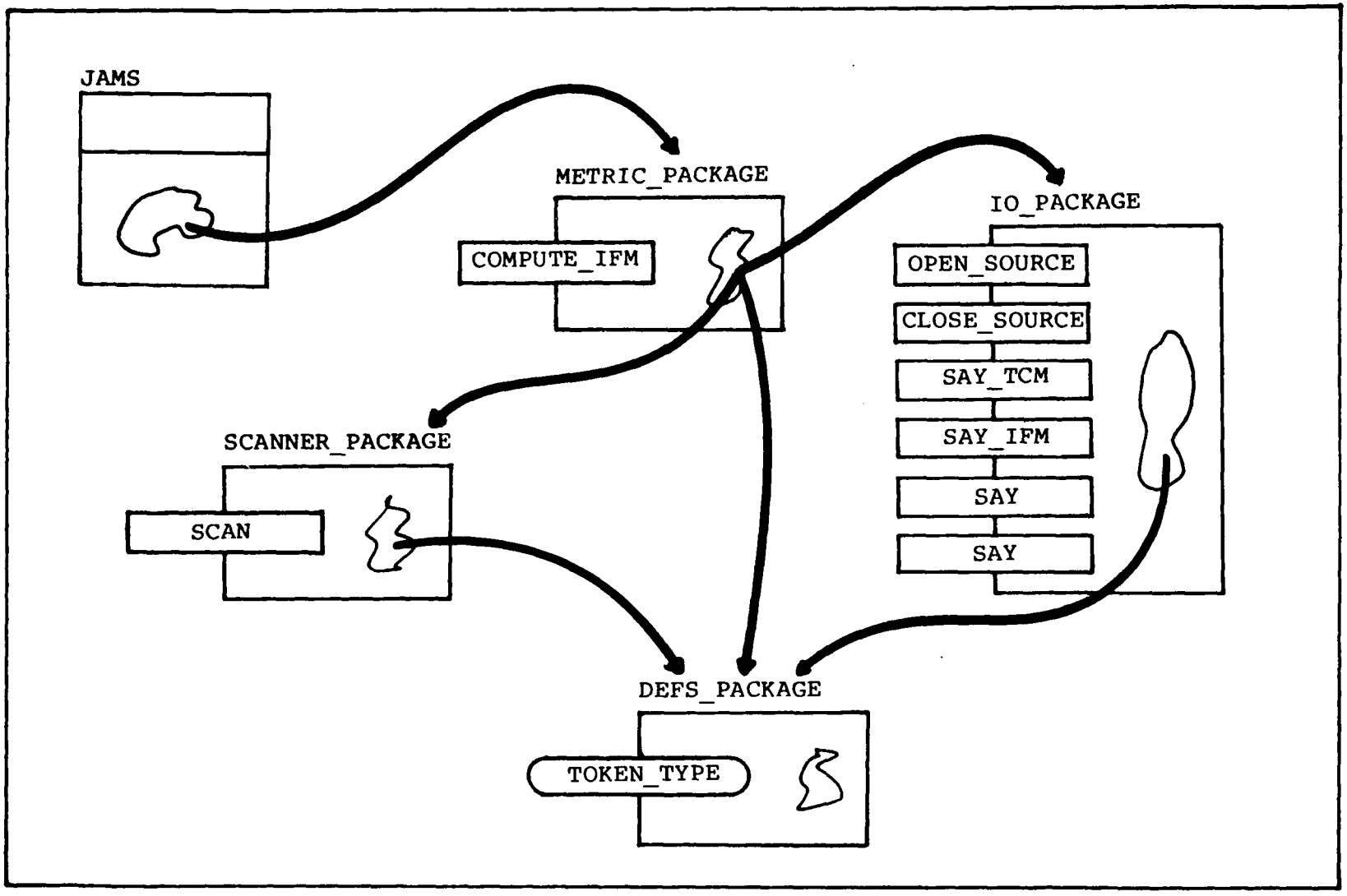

Figure 7. JAMS Program Structure

is short for JOVIAL Automated Metric System. Table 4 gives a quick description of each object. Pseudocode is presented and briefly discussed to illustrate the basic structure of each object.

The top-level object of the analyzer is a procedure called JAMS. It functions as the driver for the analyzer, and is an important part of the structure. Having three levels of abstraction minimizes the perturbations caused by individual object modification and permits the easy addition of future metrics. Pseudocode for this procedure is shown in Figure 11.

```
procedure JAMS is
begin
   print identification banner
   call COMPUTE_IFM
end JAMS
```

Figure 11. Pseudocode for the Procedure JAMS

As the program driver, JAMS outputs a banner which identifies the analyzer and version number, then calls the procedure to compute the information metric.

Table 4

JAMS Objects

Object	Description
JAMS	This object serves as the driver for the analyzer. It also helps form an abstraction which permits other metrics computations to be easily added.
METRIC_PACKAGE	This object contains the major analyzer code. It computes the metric by using the types, data structures, and operations located within itself and the other objects. Other metrics could be included in this object and simply called by the program driver, JAMS.
SCANNER_PACKAGE	This object breaks a JOVIAL (J73) source module into tokens. Placing this operation in a single object allows token definitions to be changed easily without directly affecting the rest of the code. Other tasks such as case conversion can be handled with no direct visibility from other objects.
IO_PACKAGE	This object provides communication with the user. It opens and closes the source module and displays the analysis results. Placing these operations in a separate object provides the ability to make changes to the user interface without affecting the rest of the code. Some changes that might be desirable include: adding the capability to process batches of modules, addition of software switches to tailor the analysis, or re-design of the display format.
DEFS_PACKAGE	This object is used to define a type needed by the other three objects. It has no body.

The mid-level object of the analyzer is a package called METRIC_PACKAGE. It functions as the parser and does the majority of the work in the analyzer. Only one metric is implemented in this study so one operation, COMPUTE_IFM, is provided. Pseudocode for this package is shown in Figure 12.

```
package METRIC_PACKAGE is
   procedure COMPUTE_IFM
end METRIC_PACKAGE
package body METRIC_PACKAGE is
   procedure COMPUTE_IFM is
   instantiate an object for the token count
   instantiate an object to hold the current token
   declare the doubly-linked list (DLL) type
   instantiate a DLL anchor object
   declare OUT_OF_MEMORY exception
   procedure ANALYZE -- determine length, fan-in, fan-out
   procedure PRINT_LIST -- compute and display IFM
   procedure SUM_ALL_TOKENS -- compute tokens in module
   begin -- start of COMPUTE_IFM
      form a circular DLL
      open the JOVIAL (J73) source
      loop
         get a token
         exit loop if EOF
         case token is
            when reserved word =>
               if PROGRAM or PROC then call ANALYZE
               elsif DEF then get the next token
                  if PROC then call ANALYZE
            when any other token => null
         end case
      end loop
      sum all the tokens in the module
      display the sum
      display the length, fan-in, fan-out, and IFM for
         each subroutine analyzed
      close the JOVIAL (J73) source
   exception
      when OUT_OF_MEMORY =>
         display an error message
         display the length, fan-in, fan-out, and IFM
            for each subroutine analyzed before the
            exception was raised
   end COMPUTE_IFM
end METRIC_PACKAGE
```

Figure 12. Pseudocode for METRIC_PACKAGE

Tokens from the source file are read one at a time, and if either PROGRAM or PROC is found, the procedure ANALYZE is called. Once the end of file marker (EOF) is reached, the module token sum is computed and displayed, and then the information metric for each subroutine is computed and displayed. If the user-declared exception, OUT-OF-MEMORY, is raised, the information metric is computed and displayed for the subroutines that were analyzed before this occurrence.

Several data structures are used to hold the results of intermediate computations in METRIC_PACKAGE. Since nested (to any depth) or multiple subroutines may be contained in a JOVIAL (J73) module, a doubly-linked list is used to keep the length, fan-in, and fan-out of subroutines already analyzed. During the analysis, three singly-linked lists are used to keep track of the fan-in, fan-out, and local declarations.

The procedure ANALYZE does most of the work. The code in the body of COMPUTE_IFM looks for a subroutine declaration and calls ANALYZE when one is found. Pseudocode for ANALYZE is shown in Figure 13. The name of the subroutine is read, and a new node representing the subroutine is added to the doubly-linked list. Next, the procedure GET_FORMAL_PARMS is called to search the subroutine definition formal list for fan-in and fan-out parameters. Names are added to the appropriate singly-linked list as they are found. Finally, an evaluation loop is entered. Tokens are read, and parsed with a large case statement.

```
procedure ANALYZE is
   declare the singly-linked list (SLL) type
   instantiate a SLL object for fan-in
   instantiate a SLL object for fan-out
   instantiate a SLL object for local declarations
   procedure ADD_SUBR -- add subroutine to DLL
   procedure UPDATE_SUBR -- update DLL
   procedure ADD_PARM -- add item to SLL
   procedure GET_FORMAL_PARMS -- search formal list
   function SEARCH_PARMS return BOOLEAN -- search SLL
   procedure REMOVE_PARM -- remove item from SLL
   procedure GET_ACTUAL_PARMS -- search actual list
   procedure COUNT_PARMS -- sum fan-in and fan-out
   begin -- start of ANALYZE
      get a token (it's the name of the subroutine)
      call ADD_SUBR
      call GET_FORMAL_PARMS
      loop
         get a token
         case token is
         when reserved word =>
            if BEGIN then add 1 to the BEGIN count
            elsif END then
               add 1 to the END count
               if END count = BEGIN count then
                  call COUNT_PARMS
                  call UPDATE_SUBR
                  exit the loop
               elsif END count > BEGIN count then
                  call UPDATE_SUBR
                  exit the loop
            elsif PROC then
               call COUNT_PARMS
               call UPDATE_SUBR
               call ANALYZE -- recursive call
            elsif DEF then
               get the next token
               if PROC then
                  call COUNT_PARMS
                  call UPDATE_SUBR
                  call ANALYZE -- recursive call
            . . .

         when identifier =>
            if it's not already in either the fan-in,
               fan-out, or local SLL then
               call ADD_PARM (put in local list)
         . . .

end ANALYZE
```

Figure 13. Pseudocode for the Procedure ANALYZE

One of the two low-level objects, SCANNER_PACKAGE, provides the tokens to the parser. To do this, it skips non-essential characters like blanks, spaces, tabs, and so on, and grabs the essential characters to form the tokens. Pseudocode for this package is shown in Figure 14.

A loop ensures that all non-essential characters are bypassed until a valid token is formed. Most tokens are more than one character long, so a character of state memory is needed to hold the character that has just been read while the next character is read. The data object NEXT_CHAR is used to implement this state memory. The procedure READ takes a character from either the source file (if NEXT_CHAR is empty) or NEXT_CHAR, and puts it into THIS_CHAR. Execution of the procedures INSPECT and then ADVANCE is equivalent to executing READ once. When the source file and NEXT_CHAR are both empty, an EOF token is returned. The token that a character "belongs to" is determined with a large case statement. The functions UPPER and CHECK_RESERVED are hidden within the procedure SCAN because nothing further up the abstraction needs them visible.

The other low-level object in the analyzer is the package called IO_PACKAGE. While the operations contained in this object could be included directly in METRIC_PACKAGE, it is more advantageous to have a separate object. This permits modifications to the user interface without any impact to the other objects in the analyzer. JAMS is invoked with a single command line entry like

```
package SCANNER_PACKAGE is
   function SCAN return TOKEN_TYPE
end SCANNER_PACKAGE
package body SCANNER_PACKAGE is
   instantiate an object to hold a character (THIS_CHAR)
   instantiate an object to hold a character (NEXT_CHAR)
   function SCAN return TOKEN_TYPE is
   procedure READ -- move a character into CHAR_1 from
                  -- either CHAR_2 or the source file
   procedure INSPECT -- get character from source file
   procedure ADVANCE -- move CHAR_2 into CHAR_1
   procedure BUILD_TOKEN is
   function UPPER return CHARACTER -- lower to upper
   begin -- start of BUILD_TOKEN
      call UPPER
      put a character into the TOKEN_BUFFER
   end BUILD_TOKEN
   function CHECK_RESERVED return TOKEN_TYPE is
   begin -- start of CHECK_RESERVED
      if the token in the TOKEN_BUFFER is a reserved
         JOVIAL (J73) word then
         return RESERVED_WORD
      else
         return IDENTIFIER
   end CHECK_RESERVED
   begin -- start of SCAN
      loop
      if no more characters can be read then return EOF
         call READ
         case THIS_CHAR is
         when valid name character =>
            call BUILD_TOKEN
            loop
               call INSPECT
               case NEXT_CHAR is
               when valid name character =>
                  call ADVANCE
                  call BUILD_TOKEN
               when others =>
                  if TOKEN_LENGTH = 1 then
                     return CHARACTER_LITERAL
                  else
                     return CHECK_RESERVED
               end case
            end loop
         . . .

   end SCAN
end SCANNER_PACKAGE
```

Figure 14. Pseudocode for SCANNER_PACKAGE

```
jams myprog.jov
```

where the .jov extension is used by convention and is not required by JAMS. The output is directed to the screen in a simple format. Any portion of this interface could be easily modified since these operations are encapsulated away from the rest of the analyzer objects. Pseudocode for this package is shown in Figure 15.

```
Package IO_PACKAGE is
   procedure OPEN_SOURCE
   procedure CLOSE_SOURCE
   procedure SAY_TCM
   procedure SAY_IFM
   procedure SAY    -- overloaded operator
   procedure SAY
end IO_PACKAGE
package body IO_PACKAGE is
   instantiate a new integer package (INT_IO)
   instantiate an object to hold the source file name
   instantiate an object for the source file name length
   procedure OPEN_SOURCE -- get source file name from
                         -- command line and open file
   procedure CLOSE_SOURCE -- close source file
   procedure SAY_TCM -- display module token count
   procedure SAY_IFM -- display information metric
   procedure SAY -- display a message
   procedure SAY -- display a number
end IO_PACKAGE
```

Figure 15. Pseudocode for IO_PACKAGE

Each of the operations is used by METRIC_PACKAGE to display the results of the analysis. The SAY operator is overloaded; one operation displays strings and the other displays natural numbers. Although only the SAY for strings is needed for daily JAMS operation, the SAY for numbers was used during development and was left for maintenance.

The last object, DEFS_PACKAGE, is a specification only with no package body. It simplifies the visibilities among

the mid and lower-level objects by containing a type and some objects needed by all of them. The type definition for a token (TOKEN_TYPE) is in this object. Both SCANNER_PACKAGE and METRIC_PACKAGE use the TOKEN_TYPE, the TOKEN_BUFFER, and the TOKEN_BUFFER count. Both SCANNER_PACKAGE and IO_PACKAGE use the source file name. Pseudocode for this package is shown in Figure 16.

```
package DEFS_PACKAGE is
   instantiate the TOKEN_TYPE
   instantiate an object to hold the source file name
   instantiate an object to hold the token (TOKEN_BUFFER)
   instantiate an object for the TOKEN_BUFFER character
      count
end DEFS_PACKAGE
```

Figure 16. Pseudocode for DEFS_PACKAGE

Summary

A working analyzer that computes the information metric has been implemented in JANUS/Ada. Volume 2 of this thesis contains a complete code listing. Copies may be requested from AFIT/ENG, Wright-Patterson AFB OH, 45433.

Ada can be a challenging language to use, especially on a microcomputer, but good reference books make the task much easier. The authors of several helpful books used in this study are: Amoroso and Ingargiola, Booch, Skansholm, Stanley and others, and R. R. Software.

Chapter 5 presents the analysis of some simple examples (to test specific language constructs) and also some actual F-16 flight software. Comparison of analyzer results with the sponsor's present manual process results is addressed.

V. Analyzer Verification/Validation

The construction of a working analyzer was described in Chapter 4. The rules used to build it were based on three factors: JOVIAL (J73) features affecting computation of the information metric, the data required to compute the metric, and the underlying JANUS/Ada program structure.

Now JAMS must be tested to ensure that it computes the information metric as expected. First, a few simple code samples will be examined. Next, several actual F-16 flight software modules will be analyzed. Although the results of the sponsor's manual review process will not be available for comparison with the analyzer results in this thesis, a comparison plan is presented.

Simple Examples

The first example program is shown in Figure 17. It is a main program module which illustrates these language features:

- item declarations
- table declarations
- FOR-loop and IF statements
- subroutine definitions
- subroutine function call statements
- subroutine formal parmeter list (fan-in only)
- subroutine actual parameter list (fan-out only)

Analyzer output is shown in Figure 18. As expected, the main code (which begins with the reserved word PROGRAM)

```
START
PROGRAM FINDPRIMES;
   BEGIN
   CONSTANT ITEM LIMIT U = 100;
   ITEM INDEX U = 1;
   TABLE PRIMELIST (1:20);
      ITEM PRIME U;
   FOR I : 1 BY 1 WHILE I < LIMIT AND INDEX <= 20;
      IF CHECKPRIME (I);
      BEGIN
      PRIME (INDEX) = I;
      INDEX = INDEX + 1;
      END
   END
   PROC CHECKPRIME (ARG) B;
      BEGIN
      ITEM ARG U;
      FOR I : 2 BY 1 WHILE I < ARG;
         IF (ARG / I) * I = ARG;
            BEGIN
            CHECKPRIME = FALSE;
            RETURN;
            END
         CHECKPRIME = TRUE
      END
TERM
```

Figure 17. Example Program #1 (SofTech, 1988:9-19)

```
Air Force Institute of Technology (AFIT)
JOVIAL (J73) Automated Metric System . . JAMS Version 1.0

Module Name:  findprimes.jov
Length:     116

Subroutine Name      Length    Fan-In    Fan-Out    Info Flow

FINDPRIMES               66         1          1           66
CHECKPRIME               48         1          1           48
```

Figure 18. Analyzer Output of Example Program #1

is treated as a subroutine without a formal parameter list. The other subroutine, CHECKPRIME, is defined after the main code for use within the main program module only (it is not DEF'd for external visibility). Normal analyzer output does

not show which parameters were used in the fan-in and fan-out counts, but a special test version of JAMS which does display them (plus the ones considered to be local declarations) was made. The parameters for this program were:

Subroutine	Fan-in	Fan-out	Local
FINDPRIMES	CHECKPRIME	INDEX	LIMIT INDEX PRIMELIST PRIME
CHECKPRIME	ARG	CHECKPRIME	-----

With these available, the way JAMS analyzed this program can be studied. Looking at FINDPRIMES, it may be seen that all the local declarations are either items or tables. Since no formal parameter list exists, this is expected. The table declaration is treated as an unnamed table declaration which is why the table entry item PRIME was included separately in the local list. Notice, however, that INDEX appears in the fan-out list. The statement

```
PRIME (INDEX) = 1;
```

was interpreted as a subroutine call with INDEX as fan-out rather than the data structure reference it really is. This reflects a limitation of the syntactic analysis technique. Without the ability to determine the contextual meaning of the code, JAMS cannot distinquish between a subroutine call and a reference to a data structure. Turning to CHECKPRIME, it may be seen that no local declarations are made. The item ARG is declared but only because it is a parameter in the

subroutine formal list. The assignment of a value to the name CHECKPRIME identifies the subroutine as a function.

The second example program appears in Figure 19. It is the same as the first example program, but instead of one module, it has been re-written as three modules: a compool module, a procedure module, and a main program module. This arrangement illustrates two additional language features:

- module relationship
- DEF-REF relationship

Analyzer output for the compool module of this program is shown in Figure 20. As noted in Chapter 4, the compool module has no information flow since it is only a mechanism for making items, tables, blocks, and subroutines visible to many modules. *Consequently, JAMS found no subroutines* to analyze. However, a token count was computed so at least a size metric is provided.

Analyzer output for the procedure module of this program is also shown in Figure 20. The one subroutine defined in the module, CHECKPRIME, was analyzed. Using the special version of JAMS, the fan-in, fan-out, and local declarations of the procedure module were:

Subroutine	Fan-in	Fan-out	Local
CHECKPRIME	ARG	CHECKPRIME	-----

The item ARG is declared since it is in the subroutine formal parameter list. As before, the subroutine is a function since the subroutine name is assigned a value.

```
START                                    % Compool Module %
COMPOOL CONSTRAINTS;
   CONSTANT ITEM LIMIT U = 100;
   REF PROC CHECKPRIME (ARG) B;
      ITEM ARG U;
   DEF ITEM INDEX U;
TERM


START                                    % Procedure Module %
!COMPOOL ('CONSTRAINTS')
DEF PROC CHECKPRIME (ARG) B;
   BEGIN
   ITEM ARG U;
   FOR I : 2 BY 1 WHILE I < ARG;
      IF (ARG/I) * I = ARG;
         BEGIN
         CHECKPRIME = FALSE;
         RETURN;
         END
   CHECKPRIME = TRUE;
   END
TERM


START                                    % Main Program Module %
!COMPOOL ('CONSTRAINTS');
PROGRAM FINDPRIMES;
   BEGIN
   TABLE PRIMELIST (1:20);
      ITEM PRIME U;
   INDEX = 1;
   FOR I : 1 BY 1 WHILE I < LIMIT AND INDEX <= 20;
      IF CHECKPRIME (I);
         BEGIN
         PRIME (INDEX) = I;
         INDEX = INDEX + 1;
         END
   END
TERM
```

Figure 19. Example Program #2 (SofTech, 1988:9-39 thru 9-40)

Analyzer output for the main program module of this program is shown in Figure 20, too. As in the first example program, the main code is treated as a subroutine. It is the only code in the module since the subroutine CHECKPRIME

```
Air Force Institute of Technology (AFIT)
JOVIAL (J73) Automated Metric System . . JAMS Version 1.0

Module Name:  constraints.cpl
Length:       29

No subroutines to be analyzed!


Air Force Institute of Technology (AFIT)
JOVIAL (J73) Automated Metric System . . JAMS Version 1.0

Module Name:  checkprime.jov
Length:       54

Subroutine Name      Length    Fan-In    Fan-Out    Info Flow

CHECKPRIME             48         1          1           48


Air Force Institute of Technology (AFIT)
JOVIAL (J73) Automated Metric System . . JAMS Version 1.0

Module Name:  findprimes.jov
Length:       64

Subroutine Name      Length    Fan-In    Fan-Out    Info Flow

FINDPRIMES             57         3          1          513
```

Figure 20. Analyzer Output of Example Program #2

is declared in a procedure module. The fan-in, fan-out, and local declarations of main program module were:

Subroutine	Fan-in	Fan-out	Local
FINDPRIMES	INDEX LIMIT CHECKPRIME	INDEX	PRIMELIST PRIME

A table and an item are declared. Since the main subroutine does not have a formal parameter list, these must be local declarations. The item INDEX is now declared in the compool module and must be fan-in to this module. A statement is now used to assign it the initial value of 1 rather than using

the declaration. The item LIMIT is also declared in the compool module and must be fan-in, too. As in the first example, CHECKPRIME is fan-in since it represents a returned function value. INDEX is still considered fan-out since the statement fragment PRIME (INDEX) is interpreted as a function call rather than a table entry reference.

The third example program is shown in Figure 21. It consists of four modules: two compool modules, one procedure module, and a main program module. Some important language features illustrated by this program are:

- multiple compool use
- label use
- status types
- subroutine procedure call
- subroutine formal parameters (both fan-in and fan-out)
- subroutine actual parameters (both fan-out and fan-in)

Analyzer output for both compool modules is shown in Figure 22. As in the previous example, no subroutines were analyzed since no information flow occurred. A token count is provided the information metric is not computed.

Analyzer output for the procedure module is also shown in Figure 22. The fan-in, fan-out, and local declarations of the two subroutines declared in the module were:

Subroutine	Fan-in	Fan-out	Local
GETCHANGE	ANYPRICE ANYTENDER ANYCHANGE	ANYCHANGE	-----
NEEDMORE	-----	MESSAGE	-----

```
START                                    % Compool Module %
COMPOOL MOREDEFS;
   CONSTANT ITEM NUM U 5 = 30;
   TYPE DAYS STATUS (V(SUN), V(MON), V(TUE), V(WED),
                     V(THU), V(FRI), V(SAT));
TERM


START                                    % Compool Module %
!COMPOOL ('MOREDEFS');
COMPOOL DEFINITIONS;
   DEF TABLE SALES (1 : NUM, V(SUN) : V(SAT));
      BEGIN
      ITEM PRICE F;
      ITEM TENDER F;
      ITEM CHANGE F;
      END
   REF PROC GETCHANGE (ANYPRICE, ANYTENDER, L1 :
                       ANYCHANGE);
      BEGIN
      ITEM ANYPRICE F;
      ITEM ANYTENDER F;
      ITEM ANYCHANGE F;
      LABEL L1;
      END
   REF PROC NEEDMORE ( : MESSAGE);
      ITEM MESSAGE C 20;
TERM


START                                    % Procedure Module %
!COMPOOL ('MOREDEFS');
!COMPOOL ('DEFINITIONS');
DEF PROC GETCHANGE (ANYPRICE, ANYTENDER, L1 : ANYCHANGE);
   BEGIN
   ITEM ANYPRICE F;
   ITEM ANYTENDER F;
   ITEM ANYCHANGE F;
   LABEL L1;
   ANYCHANGE = ANYTENDER - ANYPRICE;
   IF ANYCHANGE < 0.0;
      GOTO L1;
   END
DEF PROC NEEDMORE ( : MESSAGE);
   BEGIN
   ITEM MESSAGE C 20;
   MESSAGE = 'NEED MORE MONEY!';
   END
TERM
```

- continued -

Figure 21. Example Program #3

```
                        - continued -

START                                        % Main Program Module %
!COMPOOL ('MOREDEFS');
!COMPOOL ('DEFINITIONS');
PROGRAM MAKECHANGE;
   BEGIN
   ITEM NOTE C 20;
   FOR I : 1 BY 1 WHILE I < 10;
      FOR J : V(SUN) THEN NEXT (J, 1) WHILE J <= V(SAT);
         GETCHANGE (PRICE (I,J), TENDER (I,J), ERROR1 :
         CHANGE (I, J));
   STOP;
   ERROR1 : NEEDMORE ( : NOTE);
   END
TERM
```

Figure 21. Example Program #3 (SofTech, 1988:9-44 thru 9-46)

Taking the subroutine GETCHANGE first, all the item declarations were required for the names, except one, in the formal parameter list. The label L1 was not considered to be fan-in even though it was passed into the subroutine. JAMS always ignores a label since it indicates the next executable statement when a branch to it is taken, and this is control flow rather than information flow.

Next, NEEDMORE is examined. The single item declaration is required for the fan-out parameter MESSAGE. This item is simply assigned a string value and passed back to the calling subroutine.

Analyzer output for the main program module is shown in Figure 22. The fan-in, fan-out, and local declarations were:

Subroutine	Fan-in	Fan-out	Local
MAKECHANGE	CHANGE NOTE	PRICE TENDER	NOTE

```
Air Force Institute of Technology (AFIT)
JOVIAL (J73) Automated Metric System . . JAMS Version 1.0

Module Name:  moredefs.cpl
Length:       53

No procedures to be analyzed!


Air Force Institute of Technology (AFIT)
JOVIAL (J73) Automated Metric System . . JAMS Version 1.0

Module Name:  definitions.cpl
Length:       86

No procedures to be analyzed!


Air Force Institute of Technology (AFIT)
JOVIAL (J73) Automated Metric System . . JAMS Version 1.0

Module Name:  makechgproc.jov
Length:       75

Subroutine Name      Length    Fan-In    Fan-Out    Info Flow

GETCHANGE              44         3          1          396
NEEDMORE               19         0          1            0


Air Force Institute of Technology (AFIT)
JOVIAL (J73) Automated Metric System . . JAMS Version 1.0

Module Name:  makechange.jov
Length:       91

Subroutine Name      Length    Fan-In    Fan-Out    Info Flow

MAKECHANGE             79         2          2         1264
```

Figure 22. Analyzer Output for Example Program #3

One local declaration for the item NOTE was made. The other parameters were used in subroutine calls. Notice that the analyzer did not mistake indexed items like PRICE (I, J) for subroutine calls even though this is very similar to the statement that was discussed in Example Program #1. The

reason is that a single letter cannot be a variable name (required to be at least two characters in length), and JAMS ignores single letters.

Flight Software

Testing with simple examples is a good way to exercise the analyzer to see if it correctly handles specific language features that are presented in an expected manner. However, actual software is the ultimate test since language features may be used in any imaginable manner. JAMS will now be used to analyze some F-16 Block 30B Fire Control Computer (FCC) flight software. Analyzer results will be presented and briefly discussed, but FCC source code will not be shown.

Figure 23 shows the analyzer output for a FCC procedure module. Initial analysis of the module showed that pointer names were being interpreted as either fan-in or fan-out. Since a pointer simply points to data and is not, strictly speaking, data, this study made the assumption that pointer names carry no information and should not be included in fan-in or fan-out. A minor change to JAMS mostly corrected this. Pointers used in executable statements are ignored, but item

```
Air Force Institute of Technology (AFIT)
JOVIAL (J73) Automated Metric System . . JAMS Version 1.0

Module Name:  fxpuprng.jov
Length:         120

Subroutine Name      Length    Fan-In    Fan-Out    Info Flow

FXPUPRNG                 82        10          7       401800
```

Figure 23. Analyzer Output for FCC Software - #1

declarations for pointers and pointers appearing in formal or actual parameter lists are still analyzed normally.

Figure 24 shows the analyzer output for another FCC procedure module. This module revealed a deficiency in JAMS subroutine function call parameter analysis. A call similar to this

```
FIND'ROOT ( SUM'OF'SQS ( A'VECTOR ) )
```

was in the module. JAMS interpreted the parameters to be:

Fan-in	Fan-out
FIND'ROOT	SUM'OF'SQS
	A'VECTOR

This is incomplete since SUM'OF'SQS should be listed as both fan-in and fan-out since a value is returned from a function call and then exported as a call parameter for FIND'ROOT. Unfortunately, subroutine calls in the actual parameter list was not a part of the design criteria used to implement JAMS, and a simple fix could not be made.

```
Air Force Institute of Technology (AFIT)
JOVIAL (J73) Automated Metric System . . JAMS Version 1.0

Module Name:  agbrkway.jov
Length:       123

Subroutine Name      Length    Fan-In    Fan-Out    Info Flow

AGBRKWAY                91         4         12       209664
```

Figure 24. Analyzer Output for FCC Software - #2

Figure 25 shows the analyzer output for the last FCC procedure module. Initial attempts to analyze the module were unsuccessful because of arithmetic overflow errors. The

source of the problem was finally identified as numbers that were too big for the JANUS/Ada type used (LONG_INTEGER). As a simple solution, the type was changed to FLOAT. This caused the output display to look slightly different since the metric is shown in scientific notation form.

```
Air Force Institute of Technology (AFIT)
JOVIAL (J73) Automated Metric System . . JAMS Version 1.0

Module Name:  amvsim.jov
Length:     2490

Subroutine Name    Length    Fan-In    Fan-Out      Info Flow

AMVSIM               1919        62         55    2.23143E+10
AMTGT                 275        20         13    1.85900E+07
AMVINIT               240        16         15    1.38240E+07
```

Figure 25. Analyzer Output for FCC Software - #3

Figure 26 shows the analyzer output for a FCC compool module. As with any compool module, no subroutines were analyzed. However, the length indicates that this module was much larger than any of the procedure modules which were analyzed. Although size may not be the best complexity indicator, it is the only metric available and does provide a relative measure between compool modules (Harrison and others, 1982:66).

```
Air Force Institute of Technology (AFIT)
JOVIAL (J73) Automated Metric System . . JAMS Version 1.0

Module Name:  ic0comp.cpl
Length:     9349

No subroutines to be analyzed!
```

Figure 26. Analyzer Output for FCC Software - #4

Both simple example and flight software testing revealed a number of limitations of the analyzer. Some were easily corrected on the spot, but others could not be fixed without major, time-consuming changes to JAMS. Table 5 summarizes the ones that remain. Even with these limitations, however, the analyzer is useful because the information metric numbers are repeatable and comparable (among modules).

Table 5

JAMS Limitations

1. Indexed references (using valid JOVIAL (J73) names) to data structures are incorrectly interpreted as subroutine calls.
2. Index variables (single letter) are not recognized at all.
3. Pointer variables in declarative statements or a subroutine formal parameter list are recognized.
4. Subroutine calls in a subroutine actual parameter list are incorrectly analyzed.
5. A simple subroutine body (that is, only one statement without BEGIN and END) is incorrectly analyzed.
6. Subroutine names in a subroutine formal parameter list are not recognized as passed parameters.
7. DEFINE statements are incorrectly analyzed.
8. LIKE statements are incorrectly analyzed.

Comparison Plan

Once the analyzer has been constructed and checked out with sample code, it is reasonable to expect a comparison of the output with the results of the sponsor's present manual review method. Unfortunately, the opportunity to do so will not occur in time to be included in this thesis. The actual process to make the comparison will not be too difficult, so a plan for doing it is presented.

Flight software is revised periodically in what is known as "block changes." A block change may involve many modules, perhaps 200 or more. A maintainability review is performed once the block change is received. As previously discussed in Chapter 3, this is a manual process based on a closed-form questionnaire completed by a small evaluation team (AFOTECP 800-2, Vol. 3, 28 Jan 88). Ten percent of the modules are randomly selected and reviewed to form an overall indication of program maintainability (Peercy, 1981:343-346).

A number to indicate maintainability is computed for each module as well as the entire program (Telford, 1988a). Evaluation team members answer questions using the following scale:

completely agree	strongly agree	generally agree	generally disagree	strongly disagree	completely disagree

A point value is associated with each response. It ranges from 6 (completely agree) to 1 (completely disagree). The following steps are used to make the computations:

1. For each **question**, all five answers (each evaluator's response) are averaged.

2. For each **module**, the question average (computed in step 1) for each question is averaged. This provides a single number for each module.

3. For the **program**, the module average (computed in step 2) for each module is averaged. This provides a single number for the program.

Thus, the individual modules can be ranked from least to most maintainable by using the numbers produced in Step 2.

An analyzer that measures the maintainability of JOVIAL (J73) software has been designed and implemented. Because it computes the metrics automatically without human assistance, all the modules in a program rather than a small sample can be quickly analyzed and ranked.

With two methods to determine the maintainability of JOVIAL (J73) modules available, this question arises. Are the rankings computed by each method comparable?

A meaningful comparison depends on statistical methods. Specifically, inferential statistics is applicable because conclusions based on the data will be drawn and decisions will be based on them (Daniel, 1978:1). The particular form of inferential statistics used is called hypothesis testing and may be defined as

> the process of inferring from a sample whether or not to accept a certain statement about the population [Conover, 1980:1975].

A hypothesis set is formed by two statements called the null hypothesis and the alternative hypothesis. The null hypothesis is either accepted or rejected. The results of a

statistical test procedure provides the information needed to make the decision. Statistical procedures may be classified as either parametric or nonparametric. Typically, parametric methods rely on definite assumptions about the population distribution (a normal distribution curve is frequently assumed), and most use at least an interval scale data. On the other hand, nonparametric methods make less stringent assumptions about the population. (Sometimes no assumptions at all are made.) Furthermore, nonparametric methods may be applied to nominal, ordinal, interval, or ratio scale data. (Gibbons, 1985:10,11,22,23)

The Vee diagram shown in Figure 27 illustrates a statistical solution to this question. The Vee heuristic (originated by Novak and Gowin, Learning How to Learn, 1984:55-75) was designed to help focus on the question to be answered and to distinquish between the concepts involved and the methodologies used. The diagram graphically puts the question, the problem, and the solution into concrete terms. The sponsor's present method is predicated on the evaluation team being unable to analyze all the modules in a program. That's why having the ten percent (minimum) sample is so important. The analyzer developed in this thesis provides the capability of analyzing all the modules in a JOVIAL (J73) program. But does that analysis produce results similar to the results of an evaluation team? This is the problem that the focus question highlights. The events or objects that are important to this problem are shown at the bottom of the

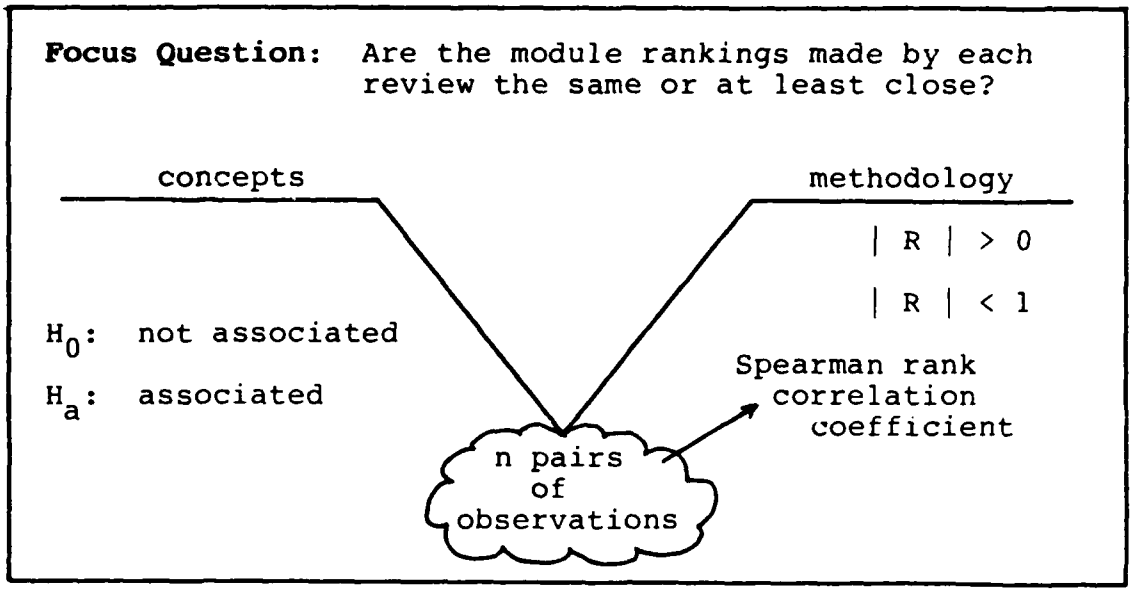

Figure 27. Vee Heuristic for Method Comparison

Vee. In this case, n pairs of similarly ranked modules are the objects. On the concept side, a null hypotheses (no association between the pairs exist) and an alternate hypothesis (some degree of association exists) are stated. On the methodology side, the Spearman rank correlation coefficient is the chosen statistic.

The Spearman rank correlation coefficient provides a relative measure of the "agreement" between paired samples which "represent the pairs of ranks of the original observations" (Gibbons, 1985:275). At least an ordinal scale must be used. The coefficient, R, may be defined as

$$R = 1 - \frac{6 \Sigma D_i^2}{n (n^2 - 1)}$$

where D_i represents the difference between any two paired ranks, and
n represents the number of sample pairs

(Gibbons, 1985:277)

R is a number between -1 ("perfect disagreement") and 1 ("perfect agreement"). Thus, when R is 0 no association is indicated and the null hypothesis must be accepted. Figure 28 illustrates a simple application of this statistic. Tables for finding the P-values for the calculated R may be found in many statistics texts. (Gibbons, 1985:273-284)

	module			
Review Method	AA	BB	CC	DD
Manual	2	4	1	3
JAMS	1	3	2	4

The differences between the ranks, D_i, are

1 1 -1 -1

and the sum of their squares is

$$\sum D_i = (1)^2 + (1)^2 + (-1)^2 + (-1)^2 = 4$$

Using the equation for R with n = 4 gives

$$R = 1 - \frac{6\ (\ 4)}{4\ (\ 15\ )} = 0.600$$

Thus, the relative measure of association between the rankings of the review methods is 0.600. From a table (Gibbons, 1985:429-431), the exact P-value is 0.208.

Figure 28. Applying the Spearman Test Statistic

If the null hypothesis is rejected, then some degree of association between the review methods is statistically indicated. This permits inference about the maintainability of the modules which were not reviewed by the evaluation team. However, if the null hypothesis cannot be rejected, no

association is statistically indicated and no inferences are possible. If a comparison finds no association, a more in-depth study will be required to determine why. Some work to gather maintenance data which could be useful for this has already been done (Peercy and others, 1987:74-77).

Several JOVIAL (J73) modules, both sample and actual, were analyzed, and the results documented in this chapter. Although the information metric computations performed by the analyzer are not exact, the numbers are reasonable and should be useful for determining maintainability. Since the data needed to compare the analyzer results with the manual method results was not available for this thesis, a plan to perform the comparison when the data becomes available was described.

VI. Conclusions and Recommendations

The sponsor of this thesis, an AFOTEC unit responsible for reviewing F-16 flight software changes, wants to improve its ability to identify software that may be difficult to maintain. Software metrics were proposed to augment the sponsor's present manual review process.

This thesis researched the use of metrics with three questions in mind:

- Do any metrics that are useful for measuring software maintainability exist?
- If some do exist, do these metrics have empirical evidence to support them?
- If empirical evidence does exist, are these metrics applicable to JOVIAL (J73)?

While many metrics have been proposed, their capability to indicate maintainability is often not addressed. This study reviewed the underlying premises of selected metrics to determine whether they were useful for measuring software maintainability. Although empirical evidence is not actual proof that a metric measures what it is intended to measure, it is the only support currently available for metrics. As a consequence, this study was concerned only with metrics that had empirical support. Any metric designed for higher-order languages (HOLs) was considered applicable to JOVIAL (J73).

A frequently-stated reason for using a metric is to improve the "quality" of software, so a definition of quality was sought. The meaning of quality varied so much, however, that it was best described through a model. Two widely-used

models were described in the literature. Both models formed a dichotomous structure with user-oriented characteristics at one end and measurement-oriented characteristics at the other.

Although maintainability could not be measured directly, complexity was found to be a strong indicator. Complexity refers to the interface between a software system and another system (machine or human). Since software is maintained by people, the interface is a human-software one (psychological complexity). Using complexity to indicate maintainability led to metrics that are algorithmic and automatable.

Only metrics that are repeatable and can be computed without human assistance were felt useful to this study. A measure must yield the same results when re-applied if comparisions are to be made. It must also be possible to mechanize the computation of the measure if a large software system is to be analyzed.

Five categories of complexity metrics were reviewed to determine whether any of them had empirical support and could be applied to JOVIAL (J73). The categories reviewed were:

- size
- control
- data
- information
- hybrid

Several individual metrics from each category were discussed. Size and control metrics were included for completeness, but

were not seriously considered for implementation. One data metric, global data usage, was selected but later rejected because a complete analysis of the entire program is needed to compute it. The information metric does not require this and was selected. Hybrid metrics were rejected because very little empirical support was available.

An analyzer that implemented the information metric was constructed using JANUS/Ada. Since Ada is a highly portable language and few JANUS/Ada library routines were used, the analyzer is fairly portable. Object-oriented techniques were used to design the analyzer although compiler theory did have an influence.

Testing was performed with both simple code examples and actual F-16 flight software. A plan to compare the metric results with the manual review results was presented for future study.

Conclusions

Constructing a tool to analyze JOVIAL (J73) proved to be a challenge. The rich variety of syntax and data structures provided by the the language plus the restrictions imposed by syntax-directed analysis caused the analyzer to have a few limitations.

Fortunately, the JANUS/Ada compiler proved to be a good system, and very little trouble was encountered with the mechanical aspects of building the analyzer. Using memory model 1 with the trim option produced an executable module of

reasonable size (less than 85K). As with standard Ada, data input and output in a JANUS/Ada program is primitive, so invocation of the analyzer and the display of results was kept as simple as possible.

Despite some limitations, the analyzer is a useful tool. It correctly analyzes the majority of JOVIAL (J73) language constructs, and always computes the information metric in a repeatable manner. Although the numbers produced are not exactly correct, they may be used to produce a rank-ordered list of JOVIAL (J73) program modules. It may be possible to infer the maintainability of modules which were not manually evaluated by comparing this ranking with a ranking produced by the sponsor's manual review process.

Recommendations

Several improvements to this analyzer could be made without too much difficulty. Listed in order of suggested implementation, these include:

1 - revising the procedure that analyzes the subroutine call to recognize subroutines as actual parameters

2 - ensuring that single letter names are properly recognized in all contexts

3 - ensuring that pointer names are ignored in all contexts

4 - ensuring that subroutines as formal parameters are recognized and not analyzed as nested subroutines

5 - ensuring that a simple subroutine body is properly recognized

Other useful improvements are possible, but only with added difficulty. These include:

6 - adding program logic to recognize and correctly analyze DEFINE'd names and their references

7 - adding program logic to correctly analyze LIKE statements

Other improvements are less critical to the computation of the metric but are important for maximum use of the analyzer. These include:

- re-designing the user interface to execute in either a single file mode or a batch file mode
- re-designing the user interface to accept software switches to tailor the analysis; output of certain test data is the primary advantage

Finally, other metric computations could be added to the analyzer. Data flow measures would be good candidates.

Summary

This study did accomplish its goals. Several measures of software maintainability were identified. Based on metric evaluation criteria, the information metric was selected and implemented. Although the analyzer which was constructed does have some limitations, it computes the metric correctly in the majority of cases. Since the metric is automated, the results are repeatable. Further, a review of an entire JOVIAL (J73) program is possible since human evaluators are not needed. This tool will improve the sponsor's ability to identify the maintainability of JOVIAL (J73) software when used in conjunction with the present manual review process.

Bibliography

Amoroso, Serafino and Giorgio Ingargiola. Ada - An Introduction to Program Design and Coding. Marshfield, Massachusetts: Pitman Publishing Inc., 1985.

Baker, Albert L. and Stuart H. Zweben. "A Comparison of Measures of Control Flow Complexity," IEEE Transactions on Software Engineering, SE-6, 6: 506-512 (November 1980).

Basili, Victor R. Quantitative Evaluation of Software Methodology. Technical Report TR-1519. College Park MD: University of Maryland, July 1985 (AD-A160202).

-----. "Quantitative Software Complexity Models: A Panel Summary," Tutorial on Models and Metrics for Software Management and Engineering. 232-233. New York: IEEE Computer Society Press, 1980.

Basili, Victor R. and Robert W. Reiter, Jr. "Evaluating Automable Measures of Software Development," Workshop on Quantitative Software Models for Reliability, Complexity, & Cost: An Assessment of the State of the Art. 107-116. New York: IEEE Publishing Services, 1979.

Belady, L. A. and M. M. Lehman. "The Characteristics of Large Systems," Program Evolution: Processes of Software Change, edited by M. M. Lehman and L. A. Belady. Orlando FL: Academic Press, Inc., 1985.

Berns, Gerald M. "Assessing Software Maintainability," Communications of the ACM, Vol. 27, No. 1: 14-23 (January 1984).

Boehm, Barry W. "Software Engineering," IEEE Transactions on Computers, Vol. C-25: 1226-1241 (December 1976).

-----. "Software and Its Impact: A Quantitative Assessment," Datamation: 48-59 (May 1973).

Boehm, Barry W. and others. "Quantitative Evaluation of Software Quality," Tutorial on Models and Metrics for Software Management and Engineering. 218-231. New York: IEEE Computer Society Press, 1980.

Boehm, Barry W. and others. Characteristics of Software Quality. Amsterdam: North-Holland Publishing Company, 1978.

Booch, Grady. Software Engineering with Ada (Second Edition). Menlo Park CA: The Benjamin/Cummings Publishing Co., Inc., 1986.

-----. Software Components with Ada: Structures, Tools, and Subsystems. Menlo Park CA: The Benjamin/Cummings Publishing Co., Inc., 1987.

Bowen, John B. "Are Current Approaches Sufficient for Measuring Software Quality?," Special Joint Issue: Performance Evaluation Review, Vol. 7, Nos. 3 & 4/ Software Engineering Notes, Vol. 3, No. 5: 148-155 (November 1978).

Bugh, Robert A. An Empirical Investigation of Control Flow Complexity Measures. A paper submitted to the graduate faculty in partial fulfillment of the requirements for the degree of master of science. Iowa State University, 1984.

Canan, James W. "The Software Crisis," Air Force Magazine: 46-52 (May 1986).

Conte and others. Software Engineering Metrics and Models. Menlo Park CA: Benjamin/Cummings Publishing Co., Inc., 1986.

Conn, Alex Paul. "Maintenance: A Key Element in Computer Requirements Definition," Proceedings of the Fourth International COMPSAC. 394-400. New York: IEEE Computer Society, October 1980.

Conover, W. J. Practical Nonparametric Statistics (Second Edition). New York: John Wiley & Sons, 1980.

Curtis, Bill. "The Measurement of Software Quality and Complexity," Software Metrics: An Analysis and Evaluation, edited by Alan Perlis and others. Cambridge, Massachusetts: The MIT Press, 1981.

-----. "In Search of Software Complexity," Workshop on Quantitative Software Models for Reliability, Complexity, & Cost: An assessment of the State of the Art. 95-106. New York: IEEE Publishing Services, 1979.

Curtis, Bill and others. "Measuring the Psychological Complexity of Software Maintenance Tasks with the Halstead and McCabe Metrics," IEEE Transactions on Software Engineering, Vol. SE-5, No. 2: 96-104 (March 1979a).

Curtis, Bill and others. "Third Time Charm: Stronger Prediction of Programmer Performance by Software Complexity Metrics," Proceedings of the Fourth International Conference on Software Engineering. 356-360. New York: IEEE Publishing Services, 1979b.

Daniel, Wayne W. Applied Nonparametric Statistics. Boston, Massachusetts: Houghton Mifflin Company, 1978.

Department of the Air Force. Software Maintainability - Evaluation Guide. AFOTECP 800-2, Volume 3. Kirtland Air Force Base, New Mexico: HQ Air Force Operational Test & Evaluation Center, 28 January 1988.

Elshoff, James L. "An Analysis of Some Commercial PL/1 Programs," IEEE Transactions on Software Engineering, Vol. SE-2, No. 2: 113-120 (June 1976).

Engimann, Bob, ITS JOVIAL (J73) Compiler Program Manager. Telephone Interview. SofTech, Inc., Fairborn OH, 24 August 1988.

Fischer, Charles N. and Richard J. LeBlanc, Jr. Crafting a Compiler. Menlo Park CA: The Benjamin/Cummings Publishing Co., Inc., 1988.

Fosdick, Lloyd D. and Leon J. Osterweil. "Data Flow Analysis in Software Reliability," Computing Surveys, Vol. 8, No. 3: 305-330 (September 1976).

Freedman, Daniel P. and Gerald M. Weinberg. "Maintenance Reviews," Techniques of Program and System Maintenance, edited by Girish Parikh. Cambridge, Massachusetts: Winthrop Publishers, Inc., 1982.

Gibbons, Jean Dickinson. Nonparametric Methods for Quantitative Analysis (Second Edition). Columbus, Ohio: American Sciences Press, Inc., 1985.

Gilb, Tom. Software Metrics. Cambridge, Massachusetts: Winthrop Publishers, Inc., 1977.

Gustafson, David A. and others. "An Analysis of Software Changes During Maintenance and Enhancement," Conference on Software Maintenance. 92-95. Washington, D. C.: IEEE Computer Society Press, 1985.

Hansen, Wilfred J. "Measurement of Program Complexity by the Pair (Cyclomatic Number, Operator Count)," SIGPLAN Notices, Vol. 13, No. 3: 29-33 (March 1978).

Harrison, Warren and others. "Applying Software Complexity Metrics to Program Maintenance," IEEE Computer Magazine: 65-79 (September 1982).

Henry, Sallie and Dennis Kafura. "Software Structure Metrics Based on Information Flow," IEEE Transactions on Software Engineering, Vol. SE-7, No. 5: 510-518 (September 1981).

Henry, Sallie and others. "On the Relationships Among Three Software Metrics," Workshop/Symposium on Measurement and Evaluation of Software Quality. 143-150. Washington, D. C.: Association for Computing Machinery, March, 1981.

Howatt, James W. A Quantitative Characterization of Control Flow Context: Software Measures for Programming Environments. PhD Dissertation. Department of Computer Science, Iowa State University, Ames, Iowa, 1985.

Hutchens, David H. Characterizing Sofware with Objective Measurements. Phd Dissertation. University of Maryland, College Park MD, 1985 (TR-1504).

Ivan, Ion and others. "Programs Complexity: Comparative Analysis, Hierarchy, Classification," ACM SIGPLAN Notices, Vol. 22, No. 4: 94-102 (April 1987).

Kafura, Dennis and Geereddy R. Reddy. "The Use of Software Complexity Metrics in Software Maintenance," IEEE Transactions on Software Engineering, Vol. SE-13, No. 3: 335-343 (March 1987).

Kearney, Joseph K. and others. "Software Complexity Measurement," Communications of the ACM, Vol. 29, No. 11: 1044-1050 (November 1986).

Lecciso, Roberto and others. "Software Metrics: A Critical Evaluation and an Application to Pascal," Microprocessing and Microprogramming, 18: 605-616 (1986).

Lehman, M. M. "The Role of Systems and Software Technology in the Fifth Generation," Program Evolution: Processes of Software Change, edited by M. M. Lehman and L. A. Belady. Orlando FL: Academic Press, Inc., 1985.

Levitin, Anany V. "How To Measure Software Size, and How Not To," Proceedings of the Software and Applications Conference. 314-318. Washington, D. C.: Computer Society Press of the IEEE, 1986.

Li, H. F. and W. K. Cheung. "An Empirical Study of Software Metrics," IEEE Transactions on Software Engineering, Vol. SE-13, No. 6: 697-708 (June 1987).

Liu, Chester C. "A Look at Software Maintenance," Techniques of Program and System Maintenance, edited by Girish Parikh. Cambridge, Massachusetts: Winthrop Publishers, Inc., 1982.

McCall, Jim A. and others. Factors in Software Quality. Volume 1 of 3. Final Technical Report, RADC-TR-77-369, August 1976 - July 1977. Contract F30602-76-C-0417. Griffiss AFB NY: Rome Air Development Center, November 1977 (AD-A049014).

McClure, Carma L. "A Model for Program Complexity Analysis," Proceedings of the Third International Conference on Software Engineering. 149-157. New York: IEEE Computer Society, 1978.

Mohanty, Siba N. "Models and Measurements for Quality Assessment of Software," Computing Surveys, Vol. 11, No. 3: 251-275 (September 1979).

Novak, Joseph D. and D. Bob Gowin. Learning How to Learn. Cambridge, Massachusetts: Cambridge University Press, 1984.

Oviedo, Enrique I. "Control Flow, Data Flow, and Program Complexity," Proceedings of the Fourth International COMPSAC. 146-152. New York: IEEE Computer Society, October 1980.

Parikh, Girish. "Introduction - The World of Software Maintenance," Techniques of Program and System Maintenance, edited by Girish Parikh. Cambridge, Massachusetts: Winthrop Publishers, Inc., 1982.

Peercy, David E. "A Software Maintainability Evaluation Methodology," IEEE Transactions on Software Engineering, SE-7, 4: 343-351 (July 1981).

Peercy, David E. and others. "Assessing Software Supportability Risk - A Minitutorial," Proceeding of the Conference on Software Maintenance. Washington, D. C.: Computer Society of the IEEE, 1987.

Pressman, Roger S. Software Engineering. New York: McGraw-Hill Book Company, 1987.

JANUS/Ada, Version 2.0.1. R. R. Software, Inc. P.O. Box 1512, Madison WI 53701, 1988.

Rabin, Michael. O. "Complexity of Computations," Communications of the ACM, Vol. 20, No. 9: 625-633 (September 1977).

Rodríguez, Volney and W. T. Tsai. "Software Metrics Interpretation Through Experimentation," Proceedings of the IEEE Computer Software and Applications Conference. Washington, D. C.: Computer Society Press of the IEEE, 1986.

R. R. Software, Inc. JANUS/Ada Development Package (D-Pak) User Manual. 8086 Version 4.2. P.O. Box 1512, Madison WI 53701, 1986.

Rubey, Raymond J. and R. Dean Hartwick. "Quantitative Measurement of Program Quality," Proceedings of the ACM National Conference. 671-677. New York: Association for Computing Machinery, 1968.

Schneidewind, Norman F. "The State of Software Maintenance," IEEE Transactions on Software Engineering, SE-13, 3: 303-310 (March 1987).

Shaw, Mary. "When is 'Good' Enough? Evaluating and Selecting Software Metrics," Software Metrics: An Analysis and Evaluation, edited by Alan Perlis and others. Cambridge, Massachusetts: The MIT Press, 1981.

Skansholm, Jan. Ada from the Beginning. Wokingham, England: Addison-Wesley Publishers Limited, 1988.

SofTech, Inc. JOVIAL (J73) Course. MIL-STD-1589C. 3100 Presidential Drive, Fairborn OH, January 1988.

-----. Computer Programming Manual for the JOVIAL (J73) Language. 2190-24.2. 3100 Presidential Drive, Fairborn OH, 10 July 1984.

Stanley, James and others. Ada - A Programmer's Guide with Microcomputer Examples. Reading, Massachusetts: Addison-Wesley Publishing Company, Inc., 1985.

Swanson, E. Burton. "The Dimensions of Maintenance," Proceedings of the Second International Conference on Software Engineering. 492-497. Long Beach CA: IEEE Computer Society, October 1976.

Telford, Captain Daniel, Deputy for Engineering and Software Analysis. Telephone interview. AFOTEC/F-16 MSIP, Edwards Air Force Base, California, 7 October 1988a.

-----. Personal interview. 17 March 1988b.

Van Verth, Patricia B. "Testing a Model of Program Quality," The Papers of the Seventeenth SIGCSE Technical Symposium on Computer Science Education. 163-172. New York: Association for Computing Machinery, 1986.

Waite, Mitchell and others. C Primer Plus. Indianapolis IN: Howard W. Sams & Company, 1987.

Weissman, Larry. "Psychological Complexity of Computer Programs: An Experimental Methodology," ACM SIGPLAN Notices, Vol. 9, No. 6: 25-36 (June 1974).

Woodward, Martin R. and others. "A Measure of Control Flow Complexity in Program Text," IEEE Transactions on Software Engineering, Vol. SE-5, No. 1: 101-106 (January 1979).

Yau, Stephen S. and James S. Collofello. "Design Stability Measures for Software Maintenance," IEEE Transactions on Software Engineering, SE-11, 9: 849-856 (September 1985).

Zelkowitz, Marvin V. "Perspectives on Software Engineering," Computing Surveys, Vol. 10, No. 2: 197-216 (June 1978).

VITA

Captain Douglas R. Tindell was born on [redacted] He attended Auburn University in Auburn, Alabama, and earned a Bachelor of Electrical Engineering degree in 1979. After receiving his commission in the U. S. Air Force, he was assigned to the 6595th Aerospace Test Group, Atlas Division, Vandenberg AFB, California where he served in various launch-related positions. His following assignment was to the 6520th Test Group, Instrumentation Division, Edwards AFB, California where he was responsible for specialized instrumentation equipment used on cruise missile F-4 chase aircraft. In 1985, he earned a Master of Public Adminstration degree from Golden Gate University. His next assignment was to the Air Force Flight Test Center, Plans and Programs Office, Edwards AFB, California where he served as a program analyst until being assigned to the Air Force Institute of Technology, School of Engineering in 1987.

[redacted]

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution unlimited
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) AFIT/GE/ENG/88D-57	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION School of Engineering 6b. OFFICE SYMBOL (If applicable) AFIT/ENG	7a. NAME OF MONITORING ORGANIZATION
6c. ADDRESS (City, State, and ZIP Code) Air Force Institute of Technology Wright-Patterson AFB OH 45433-6583	7b. ADDRESS (City, State, and ZIP Code)
8a. NAME OF FUNDING/SPONSORING ORGANIZATION 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (City, State, and ZIP Code)	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification)
MAINTENANCE METRICS FOR JOVIAL (J73) SOFTWARE

12. PERSONAL AUTHOR(S)
Douglas R. Tindell, B.E.E, M.P.A., Captain, USAF

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
MS Thesis	FROM _____ TO _____	1988 December	116

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	Maintainability, Measurement, Computer Programs
12	05		

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

Thesis Chairman: James W. Howatt, Major, USAF
Professor of Computer Systems

Approved for release in accordance with AFR 190-1
10 Jan 1989

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED
22a. NAME OF RESPONSIBLE INDIVIDUAL James W. Howatt, Major, USAF	22b. TELEPHONE (Include Area Code) 513-255-6913 22c. OFFICE SYMBOL AFIT/ENG

Block 19.

The expense of maintaining software is greater than any other phase in the life cycle. To help reduce the costs, software which may not be maintainable should be identified before being released for use. Measures of software quality, or metrics, may be able to help do this. The goal of this study was to identify measures which could indicate the maintainability of JOVIAL (J73) software, and to implement selected ones.

Maintainability cannot be measured directly, so a strong indicator, complexity, was measured instead. Five categories of complexity metrics were reviewed: size, control, data, information, and hybrid. Through an analysis of metrics from each category, the information metric was selected for implementation.

Using Ada as the implementation language, an analyzer to compute the information metric was constructed. The design was primarily object-oriented but was influenced by compiler theory. The resulting analyzer can easily incorporate new metrics or new input/output requirements.

Testing was performed using both simple code examples and actual F-16 flight software. The analyzer properly computes the information metric with few exceptions. A plan to compare the results of the analyzer with the results of the sponsor's present manual review was described.